Computational Finance

AMSTERDAM BUSINESS SCHOOL Business

Preliminaries

General Information

- My name is Simon Broda. You can find me at REC E4.27 (by appointment). Email: s.a.broda@uva.nl.
- Format of this course: 12 lectures of 2h each (2 per week), plus computer labs.
- Final grade based on a group assignment (groups of two; 40%), an indiviual assignment (35%), and a final exam (closed book, 2h; 25%).
- Additional exercises will be made available but not graded.

Material

- These lecture slides. Available on Blackboard, Github, and Microsoft Azure.
- Books:
  - Yves Hilpisch. Python for Finance: Analyze Big Financial Data. O'Reilly, 2014. ISBN 978-1-4919-4528-5 (603 pages, c. EUR 31). Code is available on Github.
  - John C. Hull. Options, Futures and Other Derivatives. 8th Edition (or later), Prentice Hall, 2012. ISBN 978-0273759072 (847 pages, c. EUR 58).
- Further reading:
  - Python documentation
  - Yves Hilpisch. Derivatives Analytics with Python. Wiley, 2015. ISBN 978-1-119-03799-6 (374 pages, c. EUR 72). Code is available on Github.
  - Python for Data Analysis. 2nd Edition, O'Reilly, 2017. ISBN 978-1-4919-5766-0 (544 pages, c. EUR 34). Code is available on Github.

Outline and Reading List

Week	Topic	Read: Hilpisch (2014)	Read: Hull (2012)
1	Introduction to Python	Chs. 2, 4 (pp. 79-95)	
2	Dealing with Data	Chs. 4 (pp. 95-108), 6, App. C	
3	Risk Measures; Plotting	Chs. 5, 10 (pp. 398-301), 11 (pp. 307-322)	Chs. 21.1, 21.2, 21.8, 22.1, 22.2
4	Binomial Trees	Ch. 8 (pp. 218-223)	Chs. 12, 14, 19, 20.1-20.5
5	Monte Carlo Methods	Ch. 10 (pp. 265-287, 290-294)	Chs. 13, 16.3, 20.6, 25.12
6	Advanced Monte Carlo Methods	Ch. 10 (pp. 287-290)	Chs. 18, 20.7, 25.7-25.12

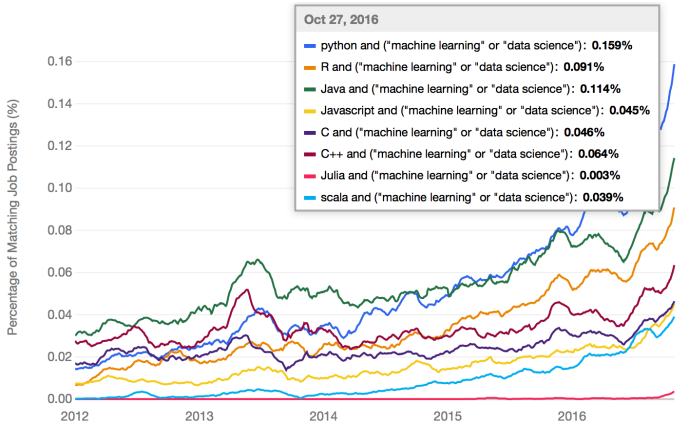
Note: Chapter numbers for Hull refer to the 8th edition. Increment by 1 for the 9th and 10th editions.

Introduction to Python

Why Python?

- General purpose programming language, unlike, e.g., Matlab®.
- High-level language with a simple syntax, interactive (*REPL*: read-eval-print loop). Hence ideal for rapid development.
- Vast array of libraries available, including for scientific computing and finance.
- Native Python is usually slower than compiled languages like C++. Alleviated by highly optimized libraries, e.g. NumPy for calculations with arrays.
- Free and open source software. Cross-platform.
- Python skills are a marketable asset: most popular language for data science.

Job Postings on Indeed.com



Source

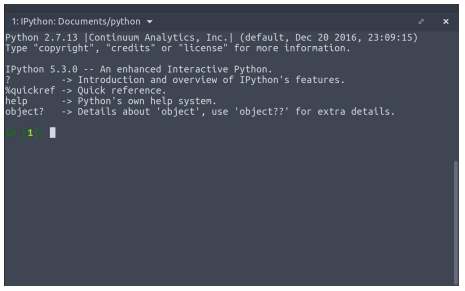
Obtaining Python

- Anaconda is a Python distribution, developed by Continuum Analytics, and specifically designed for scientific computing.
- Comes with its own package manager (conda). Many important packages (the *SciPy stack*) are pre-installed.
- Two versions: Python 2.7 and 3.6. Like the book, we will be using Python 2.7, which is still the industry standard. Most of our code should run on both with minimal adjustments.
- Obtain it from here. I recommend adding it to your PATH upon installation.
- Optional: Install the RISE plugin to allow viewing notebooks as slide shows:

In [1]:
```
#uncomment the next line to install. Note: "!" executes shell commands.
#!conda install -y -c damianavila82 rise
```


IPython Shell

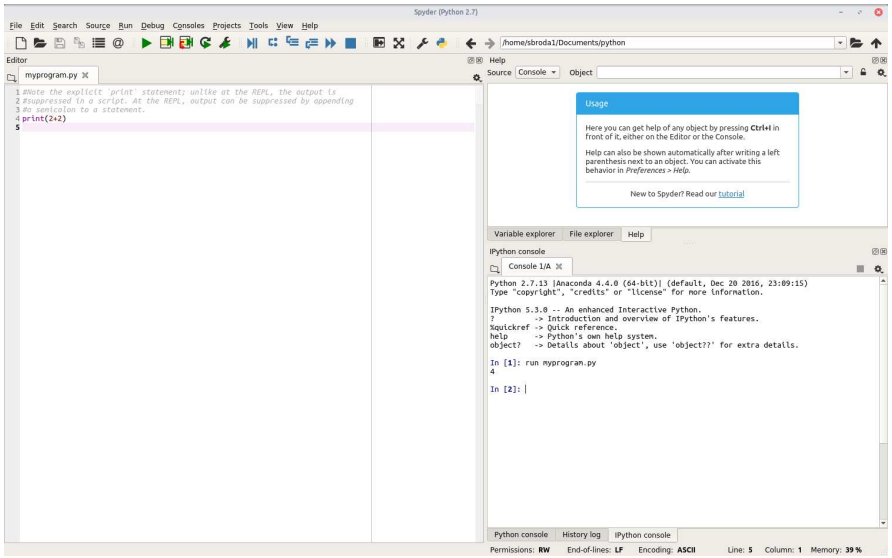
- Python features a *read-eval-print loop* (REPL) which allows you to interact with it.
- The most bare-bones method of interactive use is via the *IPython shell*:



- For now, you can treat it as a fancy calculater. Try entering 2+2. Use `quit()` or `exit()` to quit, `help()` for Python's interactive help.

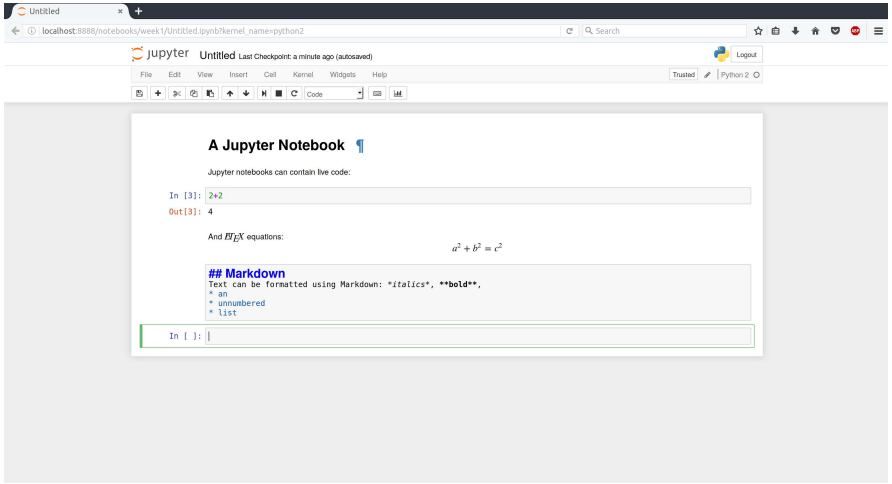
Writing Python Programs

- Apart from using it interactively, we can also write Python *programs* so we can rerun the code later.
- A Python program (called a *script* or a *module*) is just a text file, typically with the file extension `.py`.
- It contains Python commands and comments (introduced by the # character)
- To execute a program, do `run filename.py` in IPython (you may need to navigate to the right directory by using the `cd` command).
- While it is possible to code Python using just the REPL and a text editor, many people prefer to use an *integrated development environment* (IDE).
- Anaconda comes with an IDE called *Spyder* (Scientific PYthon Development EnviRonment), which integrates an editor, an IPython shell, and other useful tools.



Jupyter Notebooks

- Another option is the *Jupyter notebook* (JUlia PYThon (e) R, formerly known as IPython notebook).
- It's a web app that allows you to create documents (`*.ipynb`) that contain text (formatted in Markdown), live code, and equations (formatted in $\LaTeX$).
- In fact these very slides are based on Jupyter notebooks. You can find them on my Github page.
- The slides are also available on my Microsoft Azure page, where you can also see them as a slide show and/or run them (after cloning them; requires a free Microsoft account).



- A notebook consists of cells, each of which is either designated as Markdown (for text and equations), or as code.
- You should take a moment to familiarize yourself with the keyboard shortcuts. E.g., `enter` enters edit mode, `esc` enters command mode, `ctrl-enter` evaluates a cell, `shift-enter` evaluates a cell and selects the one below.
- Useful references:
  - Jupyter documentation;
  - Markdown cheat sheet;
  - Latex math cheat sheet.

Python Basics

Variables

- A variable is a named memory location. It is assigned using "=" (technically, "=" binds the name on the LHS to the result of the expression on the RHS).

In [2]:
```
a = 2
a = a+1  #bind the name a to the result of the expression a+1
print(a)  #show the result
```

```
3
```

In [3]:
```
a += 1  #shorthand for a = a+1
print(a)
```

```
4
```

- Variable names can be made up from letters, numbers, and the underscore. They may not start with a number. Python is case-sensitive: A is not the same as a.

Built-in Types

Attributes and Methods

- Any Python object has a *type*.
- One can use the `type` function to show the type of an object:

In [4]:
```
type(a)  #Functions take one or more inputs (in parentheses) and return an output.
```

Out[4]: int

- Objects can have attributes and methods associated with them:

In [5]:
```
a.real  #an attribute (internal variable stored inside an object)
```

Out[5]: 4

In [6]:
```
a.bit_length()  #a method (function that operates on objects of a particular type)
```

Out[6]: 3

Numeric Types

- Computers distinguish between integers and floating point numbers.
- Python integers can be arbitrary large (will use as many bits as necessary).
- Python floats are between $\pm 1.8 \cdot 10^{308}$, but are stored with just 64 bits of precision.
- Hence, not all real numbers can be represented, and floating point arithmetic is not exact:

In [7]:
```
a = 1.0; type(a)  #Note that variables can change type: a was an int before
```

Out[7]: float

In [8]:
```
a-0.9
```

Out[8]: 0.09999999999999998

Arithmetic

- The basic arithmetic operations are `+`, `-`, `*`, `/`, and `**` for exponentiation:

```
In [9]: 2*(3-1)**2
Out[9]: 8
```

- If any of the operands is a `float`, then Python will convert the others to float, too:

```
In [10]: 2*(3-1.0)**2
Out[10]: 8.0
```


- Note that `/` performs floor division in Python 2.7 (not 3.6) when both arguments are `ints`:

```
In [11]: c = 3
         c/2
Out[11]: 1
```

- We need to convert one argument to float to get the usual division:

```
In [12]: c/2.0
Out[12]: 1.5
In [13]: float(c)/2
Out[13]: 1.5
```


Booleans

- A `bool` can take one of two values: `True` or `False`.
- They are returned by *relational operators*: <, <=, >, >=, == (equality), `!=` (inequality), and can be combined using the *logical operators* `and`, `or`, and `not`.

```
In [14]: 1 <= 2 < 4
Out[14]: True
In [15]: 1 < 2 and 2 < 1
Out[15]: False
In [16]: not(1 < 2)
Out[16]: False
```


Sequence Types: Containers with Integer Indexing

Strings

- Strings hold text. They are constructed using either single or double quotes:

```
In [17]: s1 = "Python"; s2 = ' is easy.'; s1+s2  #Concatenation
Out[17]: 'Python is easy.'
```

- Strings can be indexed into:

```
In [18]: s1[0]  #Note zero-based indexing
Out[18]: 'P'
In [19]: s1[-1]  #Negative indexes count from the right:
Out[19]: 'n'
```

- We can also pick out several elements ("*slicing*"). This works for all *sequence types* (lists, NumPy arrays, ...).

In [20]:
```
s1[0:2]  #Elements 0 and 1; left endpoint is included, right endpoint excluded.
```

Out[20]:
```
'Py'
```

In [21]:
```
s1[0:6:2] #start:stop:step
```

Out[21]:
```
'Pto'
```

In [22]:
```
s1[::-1] #start and stop can be ommitted; default to 0 and len(str)
```

Out[22]:
```
'nohtyP'
```

- Strings are *immutable*:

In [23]:
```
#Wrapping this in a try block so the error doesn't break `Run all` in Jupyter.
try:
    s1[0] = "C"  #This errors.
except TypeError as e:
    print(e)
```

```
'str' object does not support item assignment
```


- Python has many useful methods for strings:

In [24]:
```
print(', '.join(filter(lambda m: callable(getattr(s1, m)) and not m.startswith("_"), dir(s1))))
```

```
capitalize, center, count, decode, encode, endswith, expandtabs, find, format, index, isalnum, isalpha, isdig
it, islower, isspace, istitle, isupper, join, ljust, lower, lstrip, partition, replace, rfind, rindex, rjust,
rpartition, rsplit, rstrip, split, splitlines, startswith, strip, swapcase, title, translate, upper, zfill
```

In [25]:
```
help(s1.upper)
```

```
Help on built-in function upper:

upper(...)
    S.upper() -> string

    Return a copy of the string S converted to uppercase.
```

In [26]:
```
(s1+s2).replace('easy','hard').upper()
```

Out[26]:
```
'PYTHON IS HARD.'
```


Lists

- Lists are indexable collections of arbitrary (though usually homogeneous) things:

In [27]:
```
list1 = [1, 2., 'hi']; print(list1)
```

```
[1, 2.0, 'hi']
```

- The function `len` returns the length of a list (or any other sequence):

In [28]:
```
len(list1)
```

Out[28]:
```
3
```

- Like strings, they support indexing, but unlike strings, they are *mutable*:

In [29]:
```
list1[2] = 42; print(list1)
```

```
[1, 2.0, 42]
```


- Note the following:

In [30]:
```
list2 = list1  #Bind the name list2 to the object list1. This does not create a copy:
list2[0] = 13
print(list1)  #list2 and list1 are the _same_ object!
```

```
[13, 2.0, 42]
```

In [31]:
```
list3 = list1[:]  #This DOES create a copy.
list3 == list1  #Tests if all elements are equal.
```

Out[31]:
```
True
```

In [32]:
```
list3 is list1  #Tests if list3 and list1 refer to the same object.
```

Out[32]:
```
False
```

In [33]:
```
list2 is list1
```

Out[33]:
```
True
```

- Lists of integers can be constructed using the `range` function:

In [34]:
```
range(1, 11, 2)  #start, stop, [,step]
```
Out[34]:
```
[1, 3, 5, 7, 9]
```

In [35]:
```
range(5)  #start and step can be ommited.
```
Out[35]:
```
[0, 1, 2, 3, 4]
```

- *List comprehensions* allow creating lists programmatically:

In [36]:
```
[x**2 for x in range(1, 10) if x > 3 and x < 7]
```
Out[36]:
```
[16, 25, 36]
```

- The `for` and `if` statements will be discussed in more detail later.

- Methods for lists:

In [37]:
```
print(', '.join(filter(lambda m: callable(getattr(list1, m)) and not m.startswith("_"), dir(list1))))
```
```
append, count, extend, index, insert, pop, remove, reverse, sort
```

In [38]:
```
list1.append(13);  #append 13 to the list `l1`
print(list1)
```
```
[13, 2.0, 42, 13]
```

In [39]:
```
list1.remove(13)  #remove first occurence of 13 from l1
print(list1)
```
```
[2.0, 42, 13]
```

- Note: Table 4-2 in the book incorrectly states that `remove[i]` removes the element at index `i`. For that, use

In [40]:
```
del(list1[0]); print(list1)
```
```
[42, 13]
```

- `del` can also be used to delete variables (technically, to unbind the variable name).

xranges

- An `xrange` is similar to a list created with `range`, but it is more memory efficient because the list elements are created on demand (*lazily*).

In [41]:
```
xrange(1, 10, 2)[3]
```
Out[41]:
```
7
```

Tuples

- A `tuple` is an immutable sequence. It is created with round brackets:

In [42]:
```
(1, 2., 'hi')
```
Out[42]:
```
(1, 2.0, 'hi')
```


Other built-in datatypes

- Other built-in datatypes include `sets` (unordered collections) and `dicts` (collections of key-value pairs). See Hilpisch (2014), pp. 92-94.

Control Flow

- Control flow refers to the order in which commands are executed within a program.
- Often we would like to alter the linear way in which commands are executed. Examples:
    1. *Conditional branch*: Code that is only evaluated if some condition is true.
    2. *Loop*: Code that is evaluated more than once.

Conditional Branch: The `if-else` statement

In [43]:
```
x = 3  #Uncomment the next line for interactive use.
#x = int(raw_input("Enter a number between 0 and 9: "))  #`raw_input` returns a string. `int` converts to inte
if x < 0:
    print("You have entered a negative number.")
elif x > 9:
    print("You have entered a number greater than 9.")
else:
    print("Thank you. You entered %s." %x)  #String interpolation.
```

```
Thank you. You entered 3.
```

- Notes:
    1. Code blocks are introduced by colons and *have* to be indented.
    2. The `if` block is executed if and only if the first condition is true
    3. The optional `elif` (short for 'else if') block is executed if and only if the first condition is false and the second one is true. There could be more than one.
    4. The optional `else` block is executed if and only if none of the others was.

`While` loops

- Similar to `if`, but jumps back to the `while` statement after the `while` block has finished.
- The `else` block is executed when the condition becomes false (not if the loop is exited through a `break` statement; see next).

In [44]:
```
x = 1  #Set this to -1 to run.
while x < 0 or x > 9:
    x = int(raw_input("Enter a number between 0 and 9: "))
    if x < 0:
        print("You have entered a negative number.")
    elif x > 9:
        print("You have entered a number greater than 9.")
else:
    print("Thank you. You entered %s." %x)
```

```
Thank you. You entered 1.
```


- Alternative implementation:

In [45]:
```
while False:  #Change to True to run.
    x = int(raw_input("Enter a number between 0 and 9: "))
    if x < 0:
        print("You have entered a negative number.")
        continue  #Skip remainder of loop body and go back to `while`.
    if x > 9:
        print("You have entered a number greater than 9.")
        continue
    print("Thank you. You entered %s." %x)
    break  #Exit innermost enclosing loop.
```

`For` Loops

- A `for` loop iterates over the elements of a sequence (e.g., a list):

In [46]:
```
for letter in "Python":
    print(letter)
```

```
P
y
t
h
o
n
```

- `letter` is called the loop variable. Every time the loop body is executed, it will in turn assume the value of each element of the sequence.

- `For` loops are typically used to execute a block of code a pre-specified number of times; `range` and `xrange` are often used in that case:

In [47]:
```
squares = []
for i in xrange(10):
    squares.append(i**2)
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Question: What does the following compute?

In [48]:
```
n = 7
f = 1
for i in xrange(n):
    f *= i+1
```


Modules

- Python's functionality is organized in *modules*.
- Some of these are part of Python's *standard library* (e.g., `math`). Others are part of *packages*, many of which come preinstalled with Anaconda (e.g., `numpy`).
- Modules need to be imported in order to make them available:

In [49]:
```
import math
math.factorial(7)
```

Out[49]:
```
5040
```

- You can use *tab completion* to discover which functions are defined by `math`: after importing, enter `math.` and press the `Tab` key. Alternatively, use dir(math):

In [50]:
```
print(', '.join(filter(lambda m: not m.startswith("_"), dir(math))))  #just so the output fits on the slide
```

```
acos, acosh, asin, asinh, atan, atan2, atanh, ceil, copysign, cos, cosh, degrees, e, erf, erfc, exp, expm1, f
abs, factorial, floor, fmod, frexp, fsum, gamma, hypot, isinf, isnan, ldexp, lgamma, log, log10, log1p, modf,
pi, pow, radians, sin, sinh, sqrt, tan, tanh, trunc
```


- Note that importing the module does not bring the functions into the *global namespace*: they need to be called as `module.function()`.
- It is possible to bring a function into the global namespace; for this, use

In [51]:
```
from math import factorial
factorial(7)
```

Out[51]:
```
5040
```

- It is even possible to import all functions from a module into the global namespace using `from math import *`, but this is frowned upon; it pollutes the namespace, which may lead to name collisions.
- *Packages* can contain several modules. They are imported the same way:

In [52]:
```
import numpy
numpy.random.rand()
```

Out[52]:
```
0.08736320389265984
```

- Optionally, you can specify a shorthand name for the imported package/module:

In [53]:
```
import numpy as np
np.sqrt(2.0)  #Note that this is not the same function as math.sqrt
```

Out[53]:
```
1.4142135623730951
```

- Conventions have evolved for the shorthands of some packages (e.g., `np` for `numpy`). Following them improves code readability.
- For the same reason, it is good practice to put your `import` statements at the beginning of your document (which I didn't do here).

Functions

Defining Functions

- User-defined functions are declared using the `def` keyword:

In [54]:
```
def mypower(x, y):  #zero or more arguments, here two
    """Compute x^y."""
    return x**y
mypower(2, 3)  #positional arguments
```

Out[54]:
```
8
```

- The *docstring* is shown by the help function:

In [55]:
```
help(mypower)
```

```
Help on function mypower in module __main__:

mypower(x, y)
    Compute x^y.
```


Several Outputs

- Functions can have more than one output argument:

In [56]:
```
def plusminus(a, b):
    return a+b, a-b
c, d = plusminus(1, 2); c, d
```

Out[56]:
```
(3, -1)
```


Keyword Arguments

- Instead of *positional arguments*, we can also pass *keyword arguments*:

In [57]:
```
mypower(y=2, x=3)
```

Out[57]:
```
9
```

- Functions can specify *default arguments*:

In [58]:
```
def mypower(x, y=2):  #default arguments have to appear at the end
    """Compute x^y."""
    return x**y
mypower(3)
```

Out[58]:
```
9
```

In [59]:
```
mypower(3, 3)
```

Out[59]:
```
27
```

Variable Scope

- Variables defined in functions are local (not visible in the calling scope):

In [60]:
```
def f():
    z = 1
f()
```

In [61]:
```
try:
    print(z)  #x is local to function f!
except NameError as e:
    print(e)
```

```
name 'z' is not defined
```


Calling Convention

- Python uses a *calling convention* known as *call by object reference*.
- This means that any modifications a function makes to its (mutable) arguments are visible to the caller (i.e., outside the function):

In [62]:
```
x = [1]  #Recall that lists are mutable.
def f(y):
    y[0] = 2  #Note: no return statement. Equivalent to `return None`.
f(x); print(x)  #Note that x has been modified in the calling scope.
```

```
[2]
```


Nested Functions

- Functions can be defined inside other functions. They will only be visible to the enclosing function.
- Nested functions can see variables defined in the enclosing function.

In [63]:
```
def mypower(x, y):
    def helper():  #No need to pass in x and y:
        return x**y  #The nested function can see them!
    a = helper()
    return a
mypower(2, 3)
```

Out[63]:
```
8
```


Advanced Material on Functions

Splatting and Slurping

- Splatting: passing the elements of a sequence into a function as positional arguments, one by one.

In [64]:
```
def mypower(x, y):
    return x**y
args = [2, 3]  #a list or a tuple
mypower(*args)  #Splat (unpack) args into mypower as positional arguments.
```

Out[64]:
```
8
```

- We can splat keyword arguments too, but we need to use a `dict` (key-value store):

In [65]:
```
kwargs={'y': 3, 'x': 2}  #a dict
mypower(**kwargs)  #splat keyword arguments
```

Out[65]:
```
8
```

- Slurping allows us to create *vararg* functions: functions that can be called with any number of positional and/or keyword arguments.

In [66]:
```
def myfunc(*myargs, **mykwargs):
    for (i, a) in enumerate(myargs): print("The %sth positional argument was %s." %(i, a))
    for a in mykwargs: print("Got keyword argument %s=%s." %(a,mykwargs[a]))
myfunc(0, 1, x=2, y=3)
```

```
The 0th positional argument was 0.
The 1th positional argument was 1.
Got keyword argument y=3.
Got keyword argument x=2.
```

- The asterisk means "collect all (remaining) positional arguments into a tuple".
- The double asterisk means "collect all (remaining) keyword arguments into a dict".

Closures

- Functions are *first class objects* in Python.
- This implies, inter alia, that functions can return other functions.
- Such functions are called *closures*, because they close around (capture) the local variables of the enclosing function.

In [67]:
```
def makemultiplier(factor):
    """Return a function that multiplies its argument by `factor`."""
    def multiplier(x):
        return x*factor
    return multiplier
timesfive = makemultiplier(5)
type(timesfive)
```

Out[67]:
```
function
```

In [68]:
```
timesfive(3)
```

Out[68]:
```
15
```


Anonymous Functions

- Anonymous functions (or *lambdas*) are functions without a name (duh...) and whose function body is a single expression.
- They are often useful for functions that are needed only once (e.g., to return from a function, or to pass to a function).
- E.g., the previous example could be written

In [69]:
```
def makemultiplier(factor):
    """Return a function that multiplies its argument by `factor`."""
    return lambda x: x*factor
timesfive=makemultiplier(5)
timesfive(3)
```

Out[69]:
```
15
```

Computational Finance

AMSTERDAM BUSINESS SCHOOL Business

Dealing with Data

More Datatypes

NumPy Arrays

- The most fundamental data type in scientific Python is `ndarray`, provided by the NumPy package (user guide).
- An array is similar to a `list`, except that
  - it can have more than one dimension;
  - its elements are homogeneous (they all have the same type).
- NumPy provides a large number of functions (*ufuncs*) that operate elementwise on arrays. This allows *vectorized* code, avoiding loops (which are slow in Python).

Constructing Arrays

- Arrays can be constructed using the `array` function which takes sequences (e.g, lists) and converts them into arrays. The data type is inferred automatically or can be specified.

```
In [2]: import numpy as np
        a = np.array([1, 2, 3, 4])
        a.dtype
Out[2]: dtype('int64')

In [3]: a = np.array([1, 2, 3, 4], dtype='float64')  #or np.array([1., 2., 3., 4.])
        a.dtype
Out[3]: dtype('float64')
```

- NumPy uses C++ data types which differ from Python's (though `float64` is equivalent to Python's `float`).

- Nested lists result in multidimensional arrays. We won't need anything beyond two-dimensional (i.e., a matrix or table).

```
In [4]: a = np.array([[1., 2.], [3., 4.]]); a
Out[4]: array([[ 1.,  2.],
               [ 3.,  4.]])

In [5]: a.ndim  #Number of dimensions.
Out[5]: 2

In [6]: a.shape #Number of rows and columns.
Out[6]: (2, 2)
```

- Other functions for creating arrays include:

```
In [7]: np.eye(3, dtype='float64')  #Identity matrix. float64 is the default dtype and can be omitted
Out[7]: array([[ 1.,  0.,  0.],
               [ 0.,  1.,  0.],
               [ 0.,  0.,  1.]])

In [8]: np.ones([2, 3])  #There's also np.zeros, and np.empty (which results in an uninitialized array).
Out[8]: array([[ 1.,  1.,  1.],
               [ 1.,  1.,  1.]])

In [9]: np.arange(0, 10, 2)  #Like range, but creates an array instead of a list.
Out[9]: array([0, 2, 4, 6, 8])

In [10]: np.linspace(0, 10, 5) #5 equally spaced points between 0 and 10
Out[10]: array([  0. ,   2.5,   5. ,   7.5,  10. ])
```


Indexing

- Indexing and slicing operations are similar to lists:

```
In [11]: a = np.array([[1., 2.], [3., 4.]])
         a[0, 0] #Element [row, column]. Equivalent to a[0][0].
Out[11]: 1.0

In [12]: b = a[:, 0]; b #Entire first column. Note that this yields a 1-dimensional array (vector), not a matrix with o
Out[12]: array([ 1.,  3.])
```

- Slicing returns *views* into the original array (unlike slicing lists):

```
In [13]: b[0] = 42

In [14]: a
Out[14]: array([[ 42.,   2.],
                [  3.,   4.]])
```


- Apart from indexing by row and column, arrays also support *Boolean* indexing:

```
In [15]: a = np.arange(10); a
Out[15]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [16]: ind = a < 5; ind
Out[16]: array([ True,  True,  True,  True,  True, False, False, False, False, False], dtype=bool)

In [17]: a[ind]
Out[17]: array([0, 1, 2, 3, 4])
```


Concatenation and Reshaping

- To combine two arrays in NumPy, use `concatenate` or `stack`:

```
In [18]: a = np.array([1, 2, 3]); b = np.array([4, 5, 6])

In [19]: c = np.concatenate([a, b]); c  #Concatenate along an existing axis.
Out[19]: array([1, 2, 3, 4, 5, 6])

In [20]: d = np.stack([a, b]); d  #Concatenate along a new axis (e.g., vectors to matrix).
Out[20]: array([[1, 2, 3],
                [4, 5, 6]])
```

- `reshape(n, m)` changes the shape of an array into `(n,m)`, taking the elements row-wise. A dimension given as `-1` will be computed automatically.

```
In [21]: d.reshape(3,-1)  #3 rows, number of columns determined automatically.
Out[21]: array([[1, 2],
                [3, 4],
                [5, 6]])
```

Arithmetic and ufuncs

- NumPy ufuncs are functions that operate elementwise:

```
In [22]: a = np.arange(1, 5); np.sqrt(a)
Out[22]: array([ 1.        ,  1.41421356,  1.73205081,  2.        ])
```

- Other useful ufuncs are `exp`, `log`, `abs`, and `sqrt`.
- Basic arithmetic on arrays works elementwise:

```
In [23]: a = np.arange(1, 5); b = np.arange(5, 9); a, b, a+b, a-b, a/b.astype(float)
Out[23]: (array([1, 2, 3, 4]),
          array([5, 6, 7, 8]),
          array([ 6,  8, 10, 12]),
          array([-4, -4, -4, -4]),
          array([ 0.2       ,  0.33333333,  0.42857143,  0.5       ]))
```


Broadcasting

- Operations between scalars and arrays are also supported:

```
In [24]: np.array([1, 2, 3, 4]) + 2
Out[24]: array([3, 4, 5, 6])
```

- This is a special case of a more general concept known as *broadcasting*, which allows operations between arrays of different shapes.
- NumPy compares the shapes of two arrays dimension-wise. It starts with the trailing dimensions, and then works its way forward. Two dimensions are compatible if
  - they are equal, or
  - one of them is 1 (or not present).
- In the latter case, the singleton dimension is "stretched" to match the larger array.

- Example:

```
In [25]: x = np.arange(6).reshape((2, 3)); x  #x has shape (2,3).
Out[25]: array([[0, 1, 2],
                [3, 4, 5]])

In [26]: m = np.mean(x, axis=0); m  #m has shape (3,).
Out[26]: array([ 1.5,  2.5,  3.5])

In [27]: x-m  #the trailing dimension matches, and m is stretched to match the 2 rows of x.
Out[27]: array([[-1.5, -1.5, -1.5],
                [ 1.5,  1.5,  1.5]])
```


- NumPy's `newaxis` feature is sometimes useful to enable broadcasting. It introduces a new dimension of length 1; e.g, it can turn a vector (1d array) into a matrix with a single row or column (2d array). Example:

```
In [28]: u = np.array([1, 2, 3])  #u has shape (3,).
         v = np.array([4, 5, 6, 7])  #v has shape (4,).
         w = u[:, np.newaxis]  #w has shape (3, 1); a matrix with 3 rows and one column.
         w*v  #(3, 1) x (4,); starting from the back, 4 and 1 are compatible, and 3 and 'missing' are too -> (3, 4).
Out[28]: array([[ 4,  5,  6,  7],
                [ 8, 10, 12, 14],
                [12, 15, 18, 21]])
```

- In this particular case, the same result could have been obtained by taking the outer product of u and v (in mathematical notation, uv'):

```
In [29]: np.outer(u, v)
Out[29]: array([[ 4,  5,  6,  7],
                [ 8, 10, 12, 14],
                [12, 15, 18, 21]])
```

Array Reductions

- *Array reductions* are operations on arrays that return scalars or lower-dimensional arrays, such as the `mean` function used above.
- They can be used to summarize information about an array, e.g., compute the standard deviation:

In [30]:
```
a = np.random.randn(300, 3)  #Create a 300x3 matrix of standard normal variates.
a.std(axis=0)  #or np.std(a, axis=0)
```

Out[30]:
```
array([ 1.01486449,  1.04777986,  1.01134656])
```

- By default, reductions operate on the *flattened* array (i.e., on all the elements). For row- or columnwise operation, the `axis` argument has to be given.
- Other useful reductions are `sum`, `median`, `min`, `max`, `argmin`, `argmax`, `any`, and `all` (see help).

Saving Arrays to Disk

- There are several ways to save an array to disk:

In [31]:
```
np.save('myfile.npy', a)  #Save `a` as a binary .npy file.
```

In [32]:
```
import os
print(os.listdir('.'))
```

```
['week5.ipynb', 'week1.ipynb', 'README.md', 'week2.ipynb', 'pdf', 'week4.ipynb', 'myfile.npy', 'week3.ipynb',
'.ipynb_checkpoints', 'img']
```

In [33]:
```
b = np.load('myfile.npy')  #Load the data into variable b.
os.remove('myfile.npy')  #Clean up.
```

In [34]:
```
np.savetxt('myfile.csv', a, delimiter=',')  #Save `a` as a CSV file (comma seperated values, can be read by MS
```

In [35]:
```
b = np.loadtxt('myfile.csv', delimiter=',')  #Load data into `b`.
os.remove('myfile.csv')
```


Pandas Dataframes

Introduction to Pandas

- `pandas` (from *p*anel *d*ata) is another fundamental package in the SciPy stack (user guide).
- It provides a number of datastructures (*series*, *dataframes*, and *panels*) designed for storing observational data, and powerful methods for manipulating (*munging*, or *wrangling*) these data.
- It is usually imported as `pd`:

In [36]:
```
import pandas as pd
```


Series

- A pandas `Series` is essentially a NumPy array with an associated index:

In [37]:
```
pop = pd.Series([5.7, 82.7, 17.0], name='Population'); pop  #The descriptive name is optional.
```

Out[37]:
```
0     5.7
1    82.7
2    17.0
Name: Population, dtype: float64
```

- The difference is that the index can be anything, not just a list of integers:

In [38]:
```
pop.index=['DK', 'DE', 'NL']
```

- The index can be used for indexing (duh...):

In [39]:
```
pop['NL']
```

Out[39]:
```
17.0
```

- NumPy's `ufuncs` preserve the index when operating on a `Series`:

```
In [40]: gdp = pd.Series([3494.898, 769.930], name='Nominal GDP in Billion USD', index=['DE', 'NL']); gdp
```

```
Out[40]: DE    3494.898
         NL     769.930
         Name: Nominal GDP in Billion USD, dtype: float64
```

```
In [41]: gdp/pop
```

```
Out[41]: DE    42.259952
         DK          NaN
         NL    45.290000
         dtype: float64
```

- One advantage of a `Series` compared to NumPy arrays is that they can handle missing data, represented as `NaN` (not a number).

Dataframes

- A `DataFrame` is a collection of `Series` with a common index (which labels the rows).

```
In [42]: data = pd.concat([gdp, pop], axis=1); data  #Concatenate two Series to a DataFrame.
```

Out[42]:

	Nominal GDP in Billion USD	Population
DE	3494.898	82.7
DK	NaN	5.7
NL	769.930	17.0

- Columns are indexed by column name:

```
In [43]: data.columns
```

```
Out[43]: Index([u'Nominal GDP in Billion USD', u'Population'], dtype='object')
```

```
In [44]: data['Population']  #data.Population works too
```

```
Out[44]: DE    82.7
         DK     5.7
         NL    17.0
         Name: Population, dtype: float64
```


- Rows are indexed with the `loc` method (note: the `ix` method listed in the book (p. 139) is deprecated):

```
In [45]: data.loc['NL']
```

```
Out[45]: Nominal GDP in Billion USD    769.93
         Population                     17.00
         Name: NL, dtype: float64
```

- Unlike arrays, dataframes can have columns with different datatypes.
- There are different ways to add columns. One is to just assign to a new column:

```
In [46]: data['Language'] = ['German', 'Danish', 'Dutch']  #Add a new column from a list.
```

- Another is to use the `join` method:

```
In [47]: s = pd.Series(['EUR', 'DKK', 'EUR', 'GBP'], index=['NL', 'DK', 'DE', 'UK'], name='Currency')
         data.join(s)  #Add a new column from a series or dataframe.
```

Out[47]:

	Nominal GDP in Billion USD	Population	Language	Currency
DE	3494.898	82.7	German	EUR
DK	NaN	5.7	Danish	DKK
NL	769.930	17.0	Dutch	EUR

- Notes:
  - The entry for 'UK' has disappeared. Pandas takes the *intersection* of indexes ('inner join') by default.
  - The returned series is a temporary object. If we want to modify `data`, we need to assign to it.

- To take the union of indexes ('outer join'), pass the keyword argument `how='outer'`:

```
In [48]: data = data.join(s, how='outer'); data  #Assignment to store the modified frame.
```

Out[48]:

	Nominal GDP in Billion USD	Population	Language	Currency
DE	3494.898	82.7	German	EUR
DK	NaN	5.7	Danish	DKK
NL	769.930	17.0	Dutch	EUR
UK	NaN	NaN	NaN	GBP

- The `join` method is in fact a convenience method that calls `pd.merge` under the hood, which is capable of more powerful SQL style operations.

- To add rows, use `loc` or `append`:

In [49]:
```
data.loc['AT'] = [386.4, 8.7, 'German', 'EUR']  #Add a row with index 'AT'.
s = pd.DataFrame([[511.0, 9.9, 'Swedish', 'SEK']], index=['SE'], columns=data.columns)
data = data.append(s)  #Add a row by appending another dataframe. May create duplicates.
data
```

Out[49]:

	Nominal GDP in Billion USD	Population	Language	Currency
DE	3494.898	82.7	German	EUR
DK	NaN	5.7	Danish	DKK
NL	769.930	17.0	Dutch	EUR
UK	NaN	NaN	NaN	GBP
AT	386.400	8.7	German	EUR
SE	511.000	9.9	Swedish	SEK

- The `dropna` method can be used to delete rows with missing values:

In [50]:
```
data = data.dropna(); data
```

Out[50]:

	Nominal GDP in Billion USD	Population	Language	Currency
DE	3494.898	82.7	German	EUR
NL	769.930	17.0	Dutch	EUR
AT	386.400	8.7	German	EUR
SE	511.000	9.9	Swedish	SEK

- Useful methods for obtaining summary information about a dataframe are `mean`, `std`, `info`, `describe`, `head`, and `tail`.

In [51]:
```
data.describe()
```

Out[51]:

	Nominal GDP in Billion USD	Population
count	4.000000	4.000000
mean	1290.557000	29.575000
std	1478.217475	35.605559
min	386.400000	8.700000
25%	479.850000	9.600000
50%	640.465000	13.450000
75%	1451.172000	33.425000
max	3494.898000	82.700000

In [52]:
```
data.head()  #Show the first few rows. data.tail shows the last few.
```

Out[52]:

	Nominal GDP in Billion USD	Population	Language	Currency
DE	3494.898	82.7	German	EUR
NL	769.930	17.0	Dutch	EUR
AT	386.400	8.7	German	EUR
SE	511.000	9.9	Swedish	SEK

- To save a dataframe to disk as a csv file, use

In [53]:
```
data.to_csv('myfile.csv')  #To_excel exists as well.
```

In [54]:
```
with open('myfile.csv', 'r') as f:
    print(f.read())
```

```
,Nominal GDP in Billion USD,Population,Language,Currency
DE,3494.898,82.7,German,EUR
NL,769.93,17.0,Dutch,EUR
AT,386.4,8.7,German,EUR
SE,511.0,9.9,Swedish,SEK
```

- To load data into a dataframe, use `pd.read_csv` (see Table 6.6 in the book):

In [55]:
```
pd.read_csv('myfile.csv', index_col=0)
```

Out[55]:

	Nominal GDP in Billion USD	Population	Language	Currency
DE	3494.898	82.7	German	EUR
NL	769.930	17.0	Dutch	EUR
AT	386.400	8.7	German	EUR
SE	511.000	9.9	Swedish	SEK

In [56]:
```
os.remove('myfile.csv')  #Clean up.
```

- Other, possibly more efficient, methods exist; see Chapter 7 of Hilpisch (2014).

Working with Time Series

Data Types

- Different data types for representing times and dates exist in Python.
- The most basic one is `datetime` from the eponymous package, and also accesible from Pandas:

In [57]:
```
pd.datetime.today()
```

Out[57]:
```
datetime.datetime(2017, 11, 19, 14, 17, 10, 832751)
```

- `datetime` objects can be created from strings using `strptime` and a format specifier:

In [58]:
```
pd.datetime.strptime('2017-03-31', '%Y-%m-%d')
```

Out[58]:
```
datetime.datetime(2017, 3, 31, 0, 0)
```

- Pandas uses `Timestamps` instead of `datetime` objects. Unlike timestamps, they store frequency and time zone information. The two can mostly be used interchangeably. See Appendix C for details.

In [59]:
```
pd.Timestamp('2017-03-31')
```

Out[59]:
```
Timestamp('2017-03-31 00:00:00')
```

- A time series is a `Series` with a special index, called a `DatetimeIndex`; essentially an array of `Timestamps`.
- It can be created using the `date_range` function; see Tables 6.2 and 6.3.

In [60]:
```
myindex = pd.date_range(end=pd.Timestamp.today(), normalize=True, periods=100, freq='B')
P = 20+np.random.randn(100).cumsum()  #Make up some share prices.
aapl = pd.Series(P, name="AAPL", index=myindex)
aapl.tail()
```

Out[60]:
```
2017-11-13    44.530210
2017-11-14    43.766867
2017-11-15    44.185038
2017-11-16    44.835533
2017-11-17    46.538283
Freq: B, Name: AAPL, dtype: float64
```


- As a convenience, Pandas allows indexing timeseries with date strings:

In [61]:
```
aapl['10/5/2017']
```

Out[61]:
```
33.100060364166268
```

In [62]:
```
aapl['10/5/2017':'10/10/2017']
```

Out[62]:
```
2017-10-05    33.100060
2017-10-06    33.773791
2017-10-09    34.235886
2017-10-10    35.551414
Freq: B, Name: AAPL, dtype: float64
```


Financial Returns

- We mostly work with returns rather than prices, because their statistical properties are more desirable (stationarity).
- There exist two types of returns: *simple returns* $R_t \equiv (P_t - P_{t-1})/P_{t-1}$, and *log returns* $r_t \equiv \log(P_t/P_{t-1}) = \log P_t - \log P_{t-1}$.
- Log returns are usually preferred, though the difference is typically small.
- To convert from prices to returns, use the `shift(k)` method, which lags by k periods (or leads if $k < 0$).

In [63]:
```
aapl=np.log(aapl)-np.log(aapl).shift(1)
aapl.tail()
```

Out[63]:
```
2017-11-13   -0.006957
2017-11-14   -0.017291
2017-11-15    0.009509
2017-11-16    0.014615
2017-11-17    0.037274
Freq: B, Name: AAPL, dtype: float64
```


- Note: for some applications (e.g., CAPM regressions), *excess returns* $r_t - r_{f,t}$ are required, where $r_{f,t}$ is the return on a "risk-free" investment.
- These are conveniently constructed as follows: suppose you have a data frame containing raw returns for a bunch of assets:

In [64]:
```
P = 20+np.random.randn(100).cumsum()  #Some more share prices.
rf = 1+np.random.randn(100)/100  #And a yield.
msft = pd.Series(P,  name="MSFT", index=myindex)
msft = np.log(msft)-np.log(msft).shift(1)
returns=pd.concat([aapl, msft], axis=1)
returns.tail()
```

Out[64]:

	AAPL	MSFT
2017-11-13	-0.006957	0.052854
2017-11-14	-0.017291	-0.083161
2017-11-15	0.009509	0.026796
2017-11-16	0.014615	0.084122
2017-11-17	0.037274	0.228075

- Then the desired operation can be expressed as

In [65]:
```
excess_returns=returns.sub(rf, axis='index')  #Subtract series rf from all columns.
```

Fetching Data

- `pandas_datareader` makes it easy to fetch data from the web (user guide).
- It is no longer included in pandas, so we need to install it.

In [66]:
```
#uncomment the next line to install.
#!conda install -y pandas-datareader
import pandas_datareader.data as web  #Not 'import pandas.io.data as web' as in the book.
```

In [67]:
```
start = pd.datetime(2010, 1, 1)
end = pd.datetime.today()
p = web.DataReader("^GSPC", 'yahoo', start, end)  #S&P500
p.tail()
```

Out[67]:

Date	Open	High	Low	Close	Adj Close	Volume
2017-11-13	2576.530029	2587.659912	2574.479980	2584.840088	2584.840088	3402930000
2017-11-14	2577.750000	2579.659912	2566.560059	2578.870117	2578.870117	3641760000
2017-11-15	2569.449951	2572.840088	2557.449951	2564.620117	2564.620117	3558890000
2017-11-16	2572.949951	2590.090088	2572.949951	2585.639893	2585.639893	3312710000
2017-11-17	2582.939941	2583.959961	2577.620117	2578.850098	2578.850098	3300160000

Regression Analysis

- Like in the book, we analyze the *leverage effect*: negative stock returns decrease the value of the equity and hence increase debt-to-equity, so the cashflow to shareholders as residual claimants becomes more risky; i.e., volatility increases.
- Hilpisch uses the VSTOXX index. Here, we use the VIX, which measures the volatility of the S&P500 based on implied volatilities from the option market.
- We already have data on the S&P500. We'll convert them to returns and do the same for the VIX. We'll store everything in a dataframe `df`.

In [68]:
```
df = pd.DataFrame()  #If in doubt, start with an empty DataFrame.
df['SP500'] = np.log(p['Adj Close']) - np.log(p['Adj Close']).shift(1)  #Make sure there's no ^ in the column
p = web.DataReader("^VIX", 'yahoo', start, end)
df['VIX'] = np.log(p['Adj Close'])-np.log(p['Adj Close']).shift(1)
df.tail()
```

Out[68]:

Date	SP500	VIX
2017-11-13	0.000983	0.018430
2017-11-14	-0.002312	0.007796
2017-11-15	-0.005541	0.124757
2017-11-16	0.008163	-0.110196
2017-11-17	-0.002629	-0.028462

- Next, we run an OLS regression of the VIX returns on those of the S&P.
- Note that this functionality has been moved from Pandas to the `statsmodels` package, so we have to use a different incantation from the one in the book.
- Also, we will use a different interface (API) which allows us to specify regressions using R-style formulas (user guide).
- We will use heteroskedasiticy and autocorrelation consistent (HAC) standard errors.

In [69]:
```
import statsmodels.formula.api as smf
model = smf.ols('VIX ~ SP500', data=df)
result = model.fit(cov_type="HAC", cov_kwds={'maxlags':5})
print(result.summary2())
```

```
                 Results: Ordinary least squares
=================================================================
Model:              OLS              Adj. R-squared:     0.653
Dependent Variable: VIX              AIC:                -6765.5638
Date:               2017-11-19 14:17 BIC:                -6754.3771
No. Observations:   1985             Log-Likelihood:     3384.8
Df Model:           1                F-statistic:        843.5
Df Residuals:       1983             Prob (F-statistic): 8.03e-155
R-squared:          0.653            Scale:              0.0019358
-----------------------------------------------------------------
               Coef.   Std.Err.     z      P>|z|    [0.025   0.975]
-----------------------------------------------------------------
Intercept      0.0024    0.0009    2.6751  0.0075   0.0006   0.0042
SP500         -6.4530    0.2222  -29.0422  0.0000  -6.8885  -6.0175
-----------------------------------------------------------------
Omnibus:              193.539       Durbin-Watson:          2.119
Prob(Omnibus):        0.000         Jarque-Bera (JB):       1143.001
Skew:                 0.238         Prob(JB):               0.000
Kurtosis:             6.687         Condition No.:          107
=================================================================
```

- Conclusion: We indeed find a significant negative effect of the index returns, confirming the existence of the leverage effect.
- Note: for a regression without an intercept, we would use `model = smf.ols('VIX ~ -1+SP500', data=df)`.
- The `result` object has useful methods and variables:

In [70]:
```
print(result.f_test('SP500=0, Intercept=0'))
```

```
<F test: F=array([[ 434.84895607]]), p=2.56661684112e-157, df_denom=1983, df_num=2>
```

In [72]:
```
result.params
```

Out[72]:
```
Intercept    0.002403
SP500       -6.453038
dtype: float64
```

Computational Finance

AMSTERDAM BUSINESS SCHOOL **Business**

Plotting Basics

- Plotting in (scientific) Python is mostly done via the `matplotlib` library (user guide), which is inspired by the plotting facilities of Matlab®.
- Its main plotting facilities reside in its `pyplot` module. It is usually imported as

In [2]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

- The second line is an ipython magic. It makes plots appear inline in the notebook.
- The `seaborn` library (user guide) provides higher-level statistical visualizations:

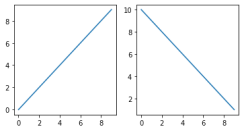
In [3]:
```
import seaborn as sns
```

- Finally, `statsmodels` is useful for QQ plots (see below):

In [4]:
```
import statsmodels.api as sm
```

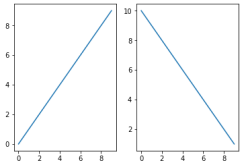

- I will only give a brief introduction to matplotlib here. However, the code for all graphs shown below is included in the notebook (though sometimes hidden in slide mode), and should be studied.
- The fundamental object in matplotlib is a `figure`, inside of which reside `subplots` (or `axes`).
- To create a new figure, add an axis, and plot to it:

In [5]:
```
#With the inline backend, these need to be in the same cell.
fig = plt.figure(figsize=(6,3))  #Create a new empty figure object. Size is optional.
ax1 = fig.add_subplot(121)  #Layout: (1x2) axes. Add one in row 1, column 1, and make it current (what plt.* c
ax2 = fig.add_subplot(122)  #Add an axes in row 1, column 2, and make it current.
ax1.plot(range(10))
ax2.plot(range(10, 0, -1));
```



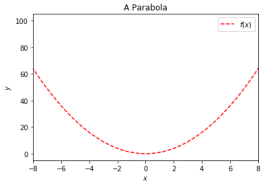
- By default, matplotlib plots into the current axis, creating one (and a figure) if needed. Using the convenience method `subplot`, this allows us to achieve the same without explicit reference to figures and axes:

In [6]:
```
plt.subplot(121)
plt.plot(range(10))
plt.subplot(122)
plt.plot(range(10, 0, -1));
```



- To plot two vectors x and y against each other:

In [7]:
```
import numpy as np
x = np.linspace(-10, 10, 100)
y = x**2
plt.plot(x,y,'r--')  #Dashed red line; see table on p. 114.
plt.xlabel('$x$')  #LaTeX equations can be included by enclosing in $$.
plt.ylabel('$y$')
plt.title('A Parabola')
plt.legend(['$f(x)$'])  #Expects a list of strings.
plt.xlim(xmin=-8, xmax=8);  #Axis limits.
#plt.savefig('filename.svg')  #Save the plot to disk.
```



Risk Measures

Introduction

- The Basel Accords mandate that financial institutions report the risk associated with their positions, so that regulators may check the adequacy of the economic capital as a buffer against market risk.
- Reporting is in the form of a *risk measure*, which condenses the risk of a position into a single number.
- Currently, the mandated measure is *Value at Risk* (VaR), but there are debates of replacing it with an alternative (*Expected Shortfall*).
- Banks are allowed to use their own, internal models for the computation of VaR, but the adequacy of these models should be *backtested*.

Value at Risk

- Consider a portfolio with value V_t and daily (simple) return R_t.
- Define the one-day loss on the portfolio as

$$\$Loss_{t+1} = -\left[V_{t+1} - V_t\right].$$

- I will distinguish between the dollar Value at Risk (an amount) and the return Value at Risk (a percentage). When unqualified, I mean the latter.
- The one-day $100p\%$ dollar Value at Risk $\$VaR^p_{t+1}$ is the loss on the portfolio that we are $100\left(1-p\right)\%$ confident will not be exceeded. The Basel committee prescribes $p = 0.01$.

- The *return Value at risk* VaR^p_{t+1} expresses $\$VaR^p_{t+1}$ as a percentage of the portfolio value:

$$VaR^p_{t+1} = \frac{\$VaR^p_{t+1}}{V_t}.$$

- Hence

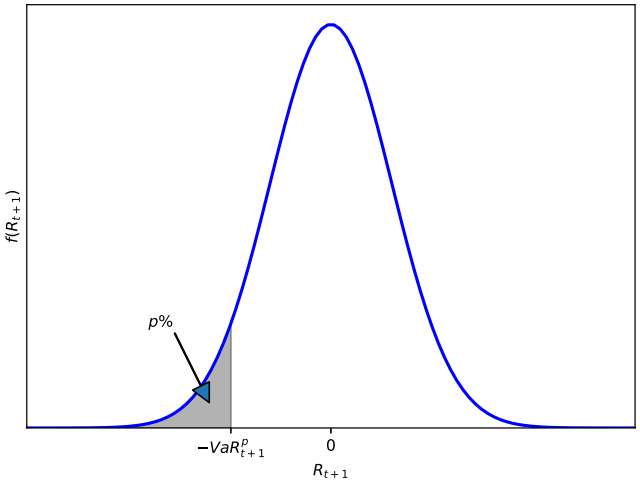
$$\Pr(R_{t+1} < -VaR^p_{t+1}) = p,$$

because

$$R_{t+1} = -\frac{\$Loss_{t+1}}{V_t}.$$

This holds approximately for log returns, too.

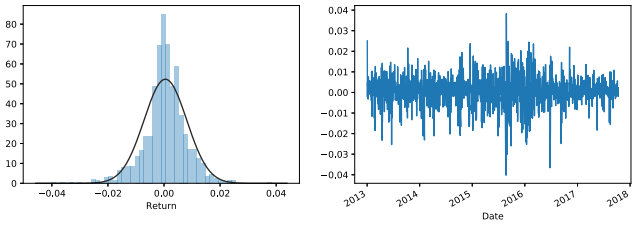
- Thus VaR^p_{t+1} is minus the $100p$th *percentile* (or minus the pth *quantile*) of the return distribution.



Asset Returns: Stylized Facts

- Stylized facts about asset returns include
  - Lack of autocorrelation;
  - Leverage effects;
  - Heavy tails of return distribution;
  - Volatility clustering.
- These need to be taken into account when creating VaR forecasts.

In [9]:
```
import pandas as pd
import pandas_datareader.data as web
import scipy.stats as stats  #The book likes to import it as `scs`.
p = web.DataReader("^GSPC", 'yahoo', start='1/1/2013', end='10/12/2017')['Adj Close']
r = np.log(p)-np.log(p).shift(1)
r.name = 'Return'
r = r[1:]  #Remove the first observation (NaN).
plt.figure(figsize=(12, 4))
plt.subplot(121)
sns.distplot(r, kde=False, fit=stats.norm)  #Histogram overlaid with a fitted normal density.
plt.subplot(122)
r.plot()  #Note that this is a pandas method! It looks prettier than plt.plot(r).
plt.savefig('img/stylizedfacts.svg')  #Save to file.
plt.close()
```

VaR Methods: Unconditional

Non-parametric: Historical Simulation

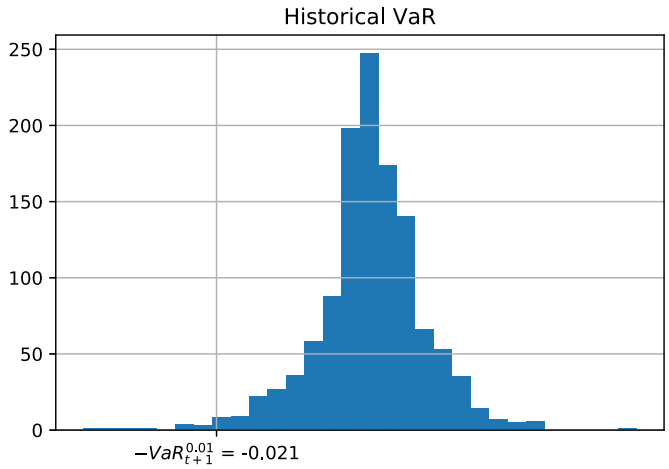
- Historical simulation assumes that the distribution of tomorrow's portfolio return is well approximated by the empirical distribution (histogram) of the past N observations $\{R_t, R_{t-1}, \ldots, R_{t+1-N}\}$.
- This is as if we draw, with replacement, from the last N returns and use this to simulate the next day's return distribution.
- The estimator of VaR is given by minus the pth *sample quantile* of the last N portfolio returns, i.e., $\widehat{VaR}^p_{t+1} = -R^N_p$, where R^N_p is the number such that $100p\%$ of the observations are smaller than it.

- In Python, we can use NumPy's `quantile` method, or the `percentile` function (or `nanpercentile` which ignores NaNs). Hilpisch uses `scoreatpercentile`, but that is is deprecated.

In [10]:
```
VaR_hist = -r.quantile(.01)  #Alternatively, VaR=np.percentile(r,1).
VaR_hist
```

Out[10]:
```
0.02131716077914799
```

In [11]:
```
ax = r.hist(bins=30)  #Another pandas method: histogram with 30 bins.
ax.set_xticks([-VaR_hist])
ax.set_xticklabels(['$-VaR_{t+1}^{0.01}$ = -%4.3f' %VaR_hist])  #4.3f means floating point with 4 digits, of w
plt.title('Historical VaR')
plt.savefig('img/var_hist.svg')
plt.close()
```

- Problem: Last year(s) of data are not necessarily representative for the next few days (because of, e.g., volatility clustering).
- Exacerbated by the fact that a large N is required to compute the 1% VaR with any degree of precision (only 1% of the data are really used).

Parametric: Normal and t Distributions

- Another simple approach is to assume $R_{t+1} \sim N(\mu, \sigma^2)$, and to estimate μ and σ^2 from historical data (for daily data, $\mu \approx 0$). The VaR is then determined from

$$\begin{aligned}\Pr\left(R_{t+1} < -VaR^p_{t+1}\right) &= \Pr\left(\frac{R_{t+1}-\mu}{\sigma} < \frac{-VaR^p_{t+1}-\mu}{\sigma}\right)\\ &= \Pr\left(z_{t+1} < \frac{-VaR^p_{t+1}-\mu}{\sigma}\right)\\ &= \Phi\left(\frac{-VaR^p_{t+1}-\mu}{\sigma}\right) = p,\end{aligned}$$

where $\Phi(z)$ is the cumulative standard normal distribution.

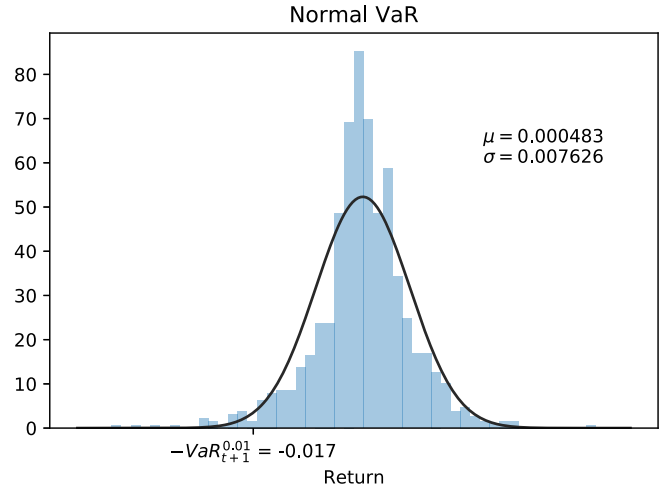
- Thus, with $\Phi^{-1}(p)$ denoting the inverse distribution function of the standard normal,

$$VaR^p_{t+1} = -\mu - \sigma\Phi^{-1}(p).$$

- Python calls $\Phi^{-1}(p)$ the *percentage point function* (ppf):

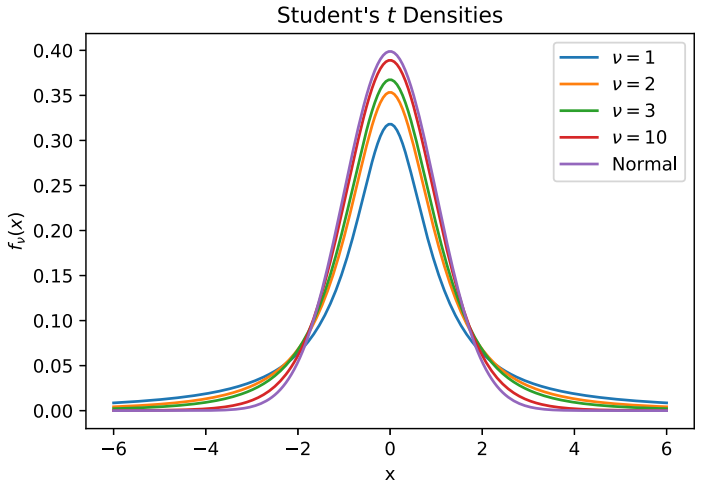
In [12]:
```
mu, sig = stats.norm.fit(r)  #Fit a normal distribution to `r`.
VaR_norm = -mu-sig*stats.norm.ppf(0.01)
VaR_norm
```

Out[12]:
```
0.01725813122968127
```

- Problems:
  - The variance of the past year(s) of data is not necessarily representative for the future.
  - Returns typically have heavier tails than the normal.
- The solution to the second point is to use another distribution. The Student's t distribution is a popular choice.

- The Student's t distribution with ν degrees of freedom, t_ν, is well known from linear regression as the distribution of t-statistics. In that context, $\nu = T - k$, where T is sample size and k the number of regressors.
- It can be generalized to allow $\nu \in \mathbb{R}_+$.
- Smaller values of ν correspond to heavier tails. As $\nu \to \infty$, we approach the $N(0, 1)$ distribution.
- It only has moments up to but not including ν:
  - The mean is finite and equal to zero if $\nu > 1$.
  - The variance is finite and equal to $\nu/(\nu - 2)$ if $\nu > 2$.
  - The excess kurtosis is finite and equal to $6/(\nu - 4)$ if $\nu > 4$.
- The distributions are symmetric around 0, so the mean and skewness are 0 if they exist.



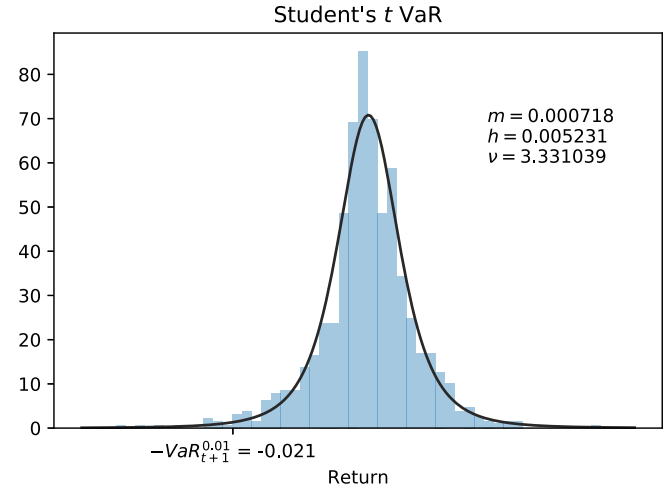
- For financial applications, we need to allow for a non-zero mean, and a variance different from $\nu/(\nu - 2)$.
- This is achieved by introducing a *location parameter* m and a *scale parameter* h. We'll write $f_\nu(x; m, h)$ for the resulting density, $F_\nu(x; m, h)$ for the distribution function, and $F_\nu^{-1}(p; m, h)$ for the percentage point function.
- Note that if $x \sim t_\nu(m, h), \nu > 2$, then $\mathbb{E}[x] = m$ and $\text{var}[x] = h^2\nu/(\nu - 2)$.
- The VaR becomes

$$VaR_{t+1}^p = -F_\nu^{-1}(p; m, h).$$

- In Python:

In [15]:
```
df, m, h = stats.t.fit(r)  #Fit a location-scale t distribution to r.
VaR_t = -stats.t.ppf(0.01, df, loc=m, scale=h)
VaR_t
```

Out[15]:
```
0.021244827811891447
```

- There are several ways to assess whether a distributional assumption is adequate.
- One is to use a *goodness of fit test*. Many such tests exist.
- Hilpisch discusses the D'Agostino-Pearson test, available as `stats.normaltest`.
- Here we use the Jarque-Bera test. The test statistic is

$$JB = N\left(S^2/6 + (K-3)/24\right),$$

where N is the sample size, and S and K are respectively the sample skewness and kurtosis.

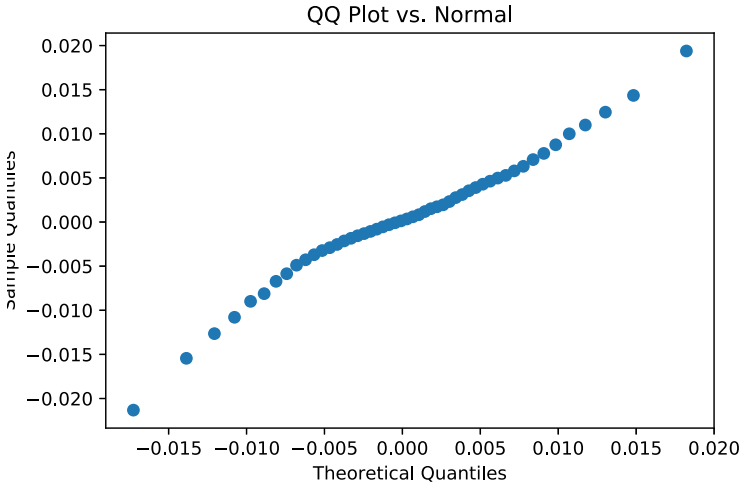
- Intuitively, it tests that the skewness and excess kurtosis are zero.
- It is distributed as χ^2_2 under the null of normality. The 5% critical value is

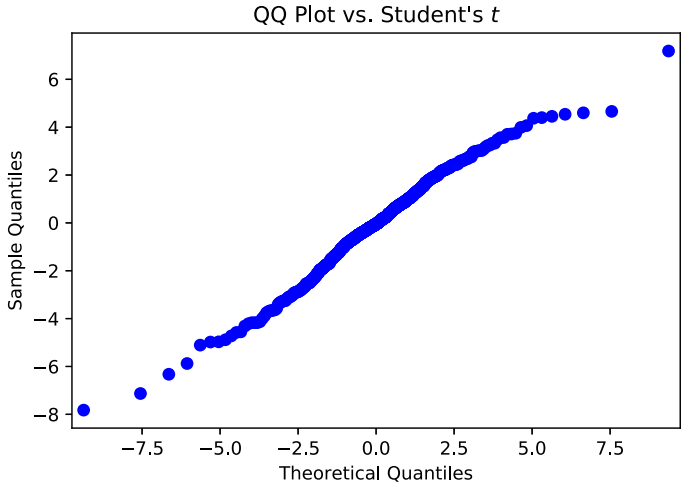
```
In [17]: stats.chi2.ppf(0.95, 2)
Out[17]: 5.9914645471079799
```

- In Python:

```
In [18]: stats.jarque_bera(r)  #Returns (JB, p-val).
Out[18]: (410.77889237295716, 0.0)
```


- Another option is to use a QQ-plot (quantile-quantile plot).
- It plots the empirical quantiles against the quantiles of a hypothesized distribution, e.g. $\Phi^{-1}(p)$ for the normal.
- If the distributional assumption is correct, then the plot should trace out the 45 degree line.





VaR Methods: Filtered

- All methods discussed so far share one drawback: they assume that the volatility is constant, at least in the estimation (and forecast) period.
- Implicitly, the Normal and Student's *t* method use the *historical volatility*:

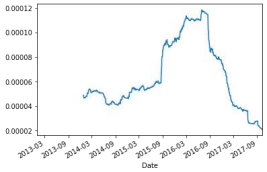
$$\sigma^2_{t+1,HIST} = \frac{1}{N}\sum_{j=0}^{N-1} R^2_{t-j}.$$

(Note: volatility usually means standard deviation, not variance. I'll be sloppy here).

- Here we assumed a zero mean, which is realistic for daily returns.
- Some adaptability is gained by choosing a smaller N such as 250 (one trading year), but there is a tradeoff because doing so decreases the sample size.
- A general solution requires a *volatility model*, which will be discussed in *Advanced Risk Management*.

- A Pandas `Series` object has a `rolling` method that can be used to construct historial volatilities for an entire series, using, at each day, the past N observations.
- The method returns a special window object that in turn has a method `var` (for variance).

In [21]:
```
sig2_hist = r.rolling(window=250).var()
sig2_hist.plot();
```



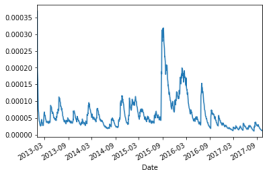
- A partial solution to the drawbacks of historical volatility is given by the RiskMetrics model, which is a special case of a more general framework known as *GARCH* models.
- The idea is to replace the equally weighted moving average used in historical volatility by an exponentially weighted moving average (EWMA):

$$\begin{aligned}\sigma^2_{t+1,EWMA} &= (1-\lambda)\sum_{j=0}^{\infty}\lambda^j R^2_{t-j} \\ &= \lambda\sigma^2_{t,EWMA} + (1-\lambda)R^2_t, \qquad 0<\lambda<1.\end{aligned}$$

- This means that observations further in the past get a smaller weight.
- Smaller λ means faster downweighting; for $\lambda \to 1$ we approach historical volatility (with an expanding window). For daily data, RiskMetrics recommends $\lambda = 0.94$.
- In practice we do not have $R_{t-\infty}$, but the second equation can be started up by an initial estimate / guess $\sigma^2_{0,EWMA}$.

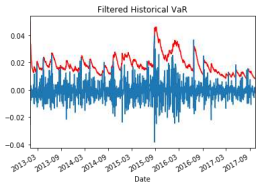
- The `ewm` (exponentially moving average) method of a Pandas `Series` can be used to achieve something similar (the exact definition is slightly different, see here).
- As before, the method returns a window object that has a `var` method.

In [22]:
```
sig2_ewma = r.ewm(alpha=0.06).var()  #alpha=(1-lambda).
sig2_ewma.plot();
```



- The idea behind a filtered VaR method is to decompose the returns as
$$R_t = \mu + \sigma_t z_t, \quad z_t \overset{\text{i.i.d}}{\sim} (0,1),$$
so that $\mathbb{E}[R_t] = \mu$ and $\text{var}[R_t] = \sigma_t^2$. In principle, μ could be time-varying as well.
- Let z_p denote the $100p\%$ percentile of
$$z_t = \frac{R_t - \mu}{\sigma_t}.$$
It can be estimated by applying any of the VaR methods above (historical, normal, or Student's t) to the *filtered* (demeaned and devolatized) returns
$$\hat{z}_t = \frac{R_t - \hat{\mu}}{\hat{\sigma}_t}.$$
- Finally, $VaR^p_{t+1} = -\mu - \sigma_{t+1} z_p$.

In [23]:
```
sig_ewma = np.sqrt(sig2_ewma)
mu = np.mean(r)
z = (r-mu)/sig_ewma
VaR_filtered_hist = -mu-sig_ewma*z.quantile(0.01)
VaR_filtered_hist.plot(color='red')
plt.plot(-r)
plt.title('Filtered Historical VaR');
```



Backtesting

- The Basel accords require that banks' internal VaR models be *backtested*.
- They recommend constructing the 1% VaR over the last 250 trading days and counting the number of *VaR exceptions* (times that losses exceeded the day's VaR figure).
- A method is said to lie in the:
    - Green zone, in case of 0-4 exceptions;
    - Yellow zone, in case of 5-9 exceptions;
    - Red zone, in case of 10 or more exceptions.
- Being in one of the latter two incurs an extra capital charge.

- A more advanced method is the *dynamic quantile* (DQ) test by Engle and Manganelli (2004).
- It is based on the *hit series*
$$I_t = \begin{cases} 1, & \text{if } r_t < -VaR^p_t, \\ 0, & \text{if } r_t > -VaR^p_t. \end{cases}$$
- If the VaR model is correctly specified, then $\mathbb{E}[I_t] = p$ (there should be $p \cdot N$ exceptions in a sample of size N, on average). This is known as the *unconditional coverage hypothesis*.
- It can be tested by regressing $I_t - p$ on an intercept and testing that it is zero.
- In addition, it is desirable that the exceptions not be correlated. This is the *independence hypothesis*. It can be tested by including lags of I_t in the regression and testing their significance.
- Jointly testing both (with an F test) tests the *conditional coverage* hypothesis.

In [24]:
```
import statsmodels.formula.api as smf
y = (r < -VaR_filtered_hist)*1  #Multiplication by 1 turns True/False into 1/0.
y.name='I'
data = pd.DataFrame(y)
model = smf.ols('I.subtract(0.01)~I.shift(1)', data=data)
res = model.fit()
print(res.summary2())
```

```
                    Results: Ordinary least squares
=================================================================
Model:              OLS              Adj. R-squared:     0.020
Dependent Variable: I.subtract(0.01) AIC:                -2069.9720
Date:               2017-11-23 14:41 BIC:                -2059.7852
No. Observations:   1204             Log-Likelihood:     1037.0
Df Model:           1                F-statistic:        25.67
Df Residuals:       1202             Prob (F-statistic): 4.68e-07
R-squared:          0.021            Scale:              0.010475
-----------------------------------------------------------------
              Coef.    Std.Err.      t      P>|t|     [0.025   0.975]
-----------------------------------------------------------------
Intercept    -0.0008     0.0030   -0.2576   0.7967   -0.0066   0.0051
I.shift(1)    0.1446     0.0285    5.0669   0.0000    0.0886   0.2006
-----------------------------------------------------------------
Omnibus:            1816.402      Durbin-Watson:         2.014
Prob(Omnibus):      0.000         Jarque-Bera (JB):      382361.558
Skew:               9.218         Prob(JB):              0.000
Kurtosis:           88.334        Condition No.:         10
=================================================================
```


- Conclusions:
  - Unconditional coverage is not rejected. This is by construction; note that $r_t \lesseqgtr -VaR_t^p \iff z_t \lesseqgtr z_p$.
  - Independence is rejected; apparently our model is dynamically mis-specified. We may need to use a more general GARCH model instead of EWMA.
- The latter finding is likely driving the rejection of the conditional coverage test:

In [25]:
```
print(res.f_test('Intercept=0, I.shift(1)=0'))
```

```
<F test: F=array([[ 12.87315967]]), p=2.93962494772e-06, df_denom=1202, df_num=2>
```

Computational Finance

AMSTERDAM BUSINESS SCHOOL **Business**

Binomial Trees

Setup and Notation

- Consider a market containing three assets: a risk-free bond with price $B_t = e^{rt}$, a stock S_t, and a (European style) derivative C_t with maturity T and payoff $C_T(S_T)$ that we wish to price.
- Split the time interval $[0, T]$ into N parts of length $\delta t = T/N$ and let $t_i = i\delta t$, $i = 0, \dots, N$, so that $t_0 = 0$ and $t_N = T$.
- Write $\{B_i, S_i, C_i, i = 0, \dots, N\}$ for the asset prices at time $t_i = i\delta t$. E.g., $C_1 \equiv C_{\delta t}$, $C_N \equiv C_T$, and $B_i = e^{r\, i\delta t}$.
- The stock price S_i either moves up to $S_{i+1}(u)$ or down to $S_{i+1}(d)$. Usually $S_{i+1}(u) = S_i u$ and $S_{i+1}(d) = S_i d$ for fixed u and d, often $u = 1/d$.

The One-Period Case: $N = 1$.

- To find C_0, construct a replicating portfolio $V_t \equiv \phi S_t + \psi B_t$ in such a way that

$$V_T(u) = \phi S_0 u + \psi B_0 e^{rT} = C(S_0 u) =: c_u,$$
$$V_T(d) = \phi S_0 d + \psi B_0 e^{rT} = C(S_0 d) =: c_d.$$

- Solving for ϕ and ψB_0 yields

$$\phi = \frac{c_u - c_d}{S_0 u - S_0 d}, \qquad \psi B_0 = e^{-rT}\left(c_u - \frac{c_u - c_d}{S_0 u - S_0 d} S_0 u\right).$$

- ϕ is known as the *hedge ratio*, or *delta* of the derivative.

- Therefore,

$$\begin{aligned} V_0 &= \phi S_0 + \psi B_0 \\ &= \frac{c_u - c_d}{u - d} + e^{-rT}\left(c_u - \frac{c_u - c_d}{u - d} u\right) \\ &= e^{-rT}\left(c_u \frac{e^{rT} - d}{u - d} + c_d \frac{u - e^{rT}}{u - d}\right) \\ &= e^{-rT}\left(c_u p + c_d [1 - p]\right). \end{aligned}$$

- In the absence of arbitrage, we must have $C_0 = V_0$, and hence

$$C_0 = e^{-rT}\left(c_u p + c_d [1 - p]\right).$$

- Interpretation: $p \in [0, 1]$, so p is a probability and C_0 is an expectation.
- p and $1 - p$ are known as *risk-neutral* probabilities. We collect these in the *risk-neutral probability measure* $\mathbb{Q}$, so that $\mathbb{Q}[u] = 1 - \mathbb{Q}[d] = p$.
- We write

$$C_0 = e^{-rT}\mathbb{E}^{\mathbb{Q}}[C_T] = e^{-rT}\left(c_u p + c_d[1-p]\right).$$

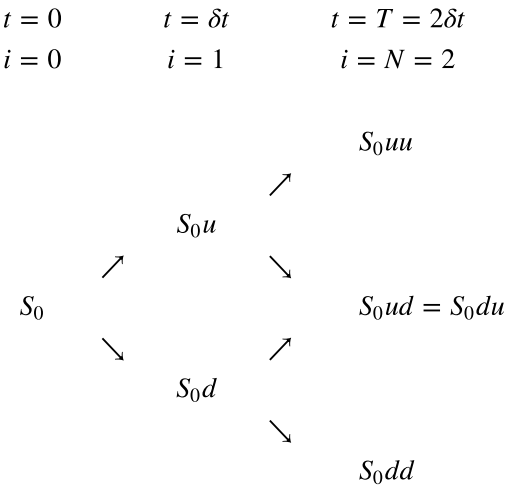
- The probabilities are called risk-neutral because if these were the true probabilities, then all assets would earn the risk-free rate. E.g., you should verify that

$$\mathbb{E}^{\mathbb{Q}}[S_T] = S_0 e^{rT}.$$

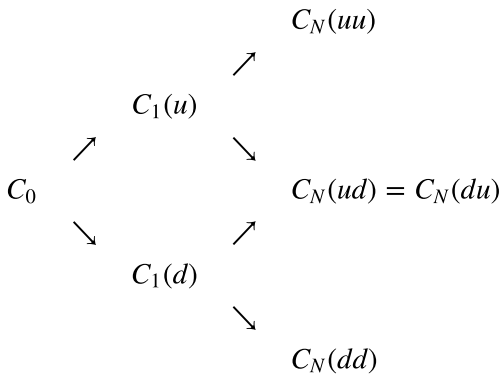
- Note that we do *not* assume that $p = \mathbb{P}[u]$. The actual probability $\mathbb{P}[u]$ is *irrelevant* for the value C_0 of the derivative (as long as it is not zero or one).

The N-Period Case

- Next, consider a two-period model ($N = 2$):



- This stock price tree is *recombinant*: an up move followed by a down move leads to the same value as a down move followed by an up move. This is a consequence of u and d being fixed and independent of the price.
- Advantage: the number of nodes remains manageable ($N + 1$ at the Nth step, rather than 2^N).
- This leads to a derivative price tree that is also recombinant. Given a recombinant stock price tree, this follows from the fact that C_N only depends on S_N.
- Path-dependent derivatives where $C_N = C(S_i, i \leq N)$ may lead to non-recombinant trees.



- Only the payoffs $C_N(uu)$, $C_N(ud)$ and $C_N(dd)$ are known, and we wish to obtain C_0, $C_1(u)$ and $C_1(d)$.
- At time $t = \delta t$ (after one step), we know whether the stock has gone up or down.
- If it has gone up, then only the branch from $C_1(u)$ to $C_N(uu)$ or $C(ud)$ is relevant.
- Since this is just a binary model, we can price $C_1(u)$ (and $C_1(d)$) by no-arbitrage:

$$C_1(u) = e^{-r\,\delta t}\left[C_N(uu)p + C_N(ud)(1-p)\right] = e^{-r\,\delta t}\mathbb{E}^{\mathbb{Q}}\left[C_N | S_1 = S_0 u\right],$$
$$C_1(d) = e^{-r\,\delta t}\left[C_N(du)p + C_N(dd)(1-p)\right] = e^{-r\,\delta t}\mathbb{E}^{\mathbb{Q}}\left[C_N | S_1 = S_0 d\right].$$

- Recall that $p = \dfrac{e^{r\,\delta t} - d}{u - d}$; in general the risk-neutral probability might depend on S_1, but in this case it doesn't, because r, u and d are the same at each step.

- The values $C_1(u)$ and $C_1(d)$ are the market prices (under the no-arbitrage condition), so the derivative can be sold at this price at time $t = \delta t$, depending on whether the stock goes up or down.
- Therefore, at time $t = 0$ we know that the two possible payoffs in the next period are $C_1(u)$ and $C_1(d)$, and so

$$\begin{aligned} C_0 &= e^{-r\,\delta t}\left[C_1(u)p + C_1(d)(1-p)\right] \\ &= e^{-rT}\left[C_N(uu)p^2 + C_N(ud)[p(1-p) + (1-p)p] \right. \\ &\qquad \left. + C_N(dd)(1-p)^2\right] \\ &= e^{-rT}\mathbb{E}^{\mathbb{Q}}\left[C_N\right]. \end{aligned}$$

- In the N-period case, denote by $\mathcal{F}_t$ the information at time t, i.e., whether the stock went up or down at each $s \leq t$. Then, at each step in the tree,
$$C_t = e^{-r\delta t}\mathbb{E}^{\mathbb{Q}}[C_{t+\delta t}|\mathcal{F}_t].$$
- Starting at C_T, this can be solved backwards until one arrives at the price at $t = 0$.
- At every step in the tree, we have that
$$C_t = e^{-r(T-t)}\mathbb{E}^{\mathbb{Q}}[C_T|\mathcal{F}_t],$$
and in particular
$$C_0 = e^{-rT}\mathbb{E}^{\mathbb{Q}}[C_T].$$
- This is known as the *risk neutral pricing formula*: the price of an attainable European claim equals the expected discounted payoff, but where expectations are under a set of risk-neutral probabilities $\mathbb{Q}$.

- It is worth noting that the hedging strategy is dynamic: let ϕ_{i+1} and ψ_{i+1} denote the number of shares and cash bonds held from t_i till t_{i+1}.
- The single-period binary model implies
$$\phi_{i+1} = \frac{C_{i+1}(u) - C_{i+1}(d)}{S_{i+1}(u) - S_{i+1}(d)}.$$
- Between t_i and t_{i+1}, the value changes from V_i to $\phi_{i+1}S_{i+1} + \psi_{i+1}B_{i+1}$, after which rebalancing occurs.
- The strategy is *replicating*: after N steps, the value is $V_N = \phi_N S_N + \psi_N B_N = C_N$.
- It can also be verified to be *self-financing*:
$$V_i = \phi_i S_i + \psi_i B_i = \phi_{i+1} S_i + \psi_{i+1} B_i,$$
which may be rewritten as
$$V_{i+1} - V_i = \phi_{i+1}(S_{i+1} - S_i) + \psi_{i+1}(B_{i+1} - B_i).$$
- Thus, a dynamic strategy allows us to hedge against more than two states at time T with only two assets.

Martingales and the FTAP

- A sequence of random variables such as $\{S_i\}_{i\geq 0}$ is called a *stochastic process*.
- Observe that under $\mathbb{Q}$,
$$\mathbb{E}^{\mathbb{Q}}[S_{i+1}|\mathcal{F}_i] = S_i\Big(up + d(1-p)\Big) = S_i e^{r\delta t}.$$
- Define the *discounted stock price process* $\tilde{S}_i = S_i e^{-ir\delta t}$. Then
$$\mathbb{E}^{\mathbb{Q}}\left[\tilde{S}_{i+1}|\mathcal{F}_i\right] = S_i e^{r\delta t} e^{-(i+1)r\delta t} = S_i e^{-ir\delta t} = \tilde{S}_i.$$
This is the defining property of a *martingale*. Hence, the risk-neutral measure is also called a *martingale measure*.
- $\mathbb{Q}$ and $\mathbb{P}$ are *equivalent* if $\mathbb{Q}[A] = 0 \iff \mathbb{P}[A] = 0$.
- *Fundamental Theorem of Asset Pricing*: if (and only if) the market is arbitrage free, then there exists an equivalent martingale measure $\mathbb{Q}$ under which discounted stock prices are martingales, and the risk neutral pricing formula holds. $\mathbb{Q}$ is unique if the market is complete.

Tree Calibration

- We are given S_0, T (measured in years), and the function $C_T = C(S_T)$; for a European call, $C(S_T) = \max\{(S_T - K), 0\}$.
- We have to choose the number N of steps, and hence $\delta t = T/N$. This involves a trade-off between computational burden and accuracy.
- $r = \log(1+R)$, where R is the current value (per annum) of a suitable risk-free interest rate (e.g. LIBOR) over the holding period of the option.
- u and d are chosen to match the stock price volatility: under $\mathbb{Q}$,
$$R_{i+1} \equiv \log(S_{i+1}/S_i) = \begin{cases} \log u & \text{with probability } p, \\ \log d = -\log u & \text{with probability } 1-p. \end{cases}$$

- Thus,
$$\mathbb{E}^{\mathbb{Q}}[R_{i+1}] = (2p-1)\log u, \quad \text{and}$$
$$\sigma^2\delta t := \mathrm{var}^{\mathbb{Q}}(R_{i+1}) = (\log u)^2\left[1 - (2p-1)^2\right] \approx (\log u)^2.$$
- Hence we choose
$$u = e^{\sigma\sqrt{\delta t}}, \qquad d = 1/u = e^{-\sigma\sqrt{\delta t}}.$$
- Possible estimates for σ:
  - Annualized historical volatility (see last week):
$$\sigma = \sqrt{252}\sigma_{t,HIST}$$
  - Implied volatility: the value of σ that equates model price and market price (see later).

Binomial Trees in Python

- We will look at several Python implementations and compare their speed.
- The first implementation is a "loopy" version that could be written in a similar way in most imperative programming languages.

In [1]:
```
import numpy as np
def calltree(S0, K, T, r, sigma, N):
    """
    European call price based on an N-step binomial tree.
    """
    deltaT = T/float(N)
    u = np.exp(sigma * np.sqrt(deltaT))
    d = 1/u
    p = (np.exp(r*deltaT) - d)/(u-d)
    C = np.zeros((N+1, N+1))
    S = np.zeros((N+1, N+1))
    piu = np.exp(-r*deltaT)*p
    pid = np.exp(-r*deltaT)*(1-p)
    for i in xrange(N+1):
        for j in xrange(i, N+1):
            S[i, j] = S0 * u**j * d**(2*i)  #or S0 * u**(j-i) * d**(i)
    for i in xrange(N+1):
        C[i, N] = max(0, S[i, N]-K)
    for j in xrange(N-1, -1, -1):
        for i in xrange(j+1):
            C[i, j] = piu * C[i, j+1] + pid * C[i+1, j+1]
    return  C[0, 0]
```


- Let's see if it works:

In [2]:
```
S0=11.; K=10.; T=3/12.; r=.02; sigma=.3; N=500;
calltree(S0, K, T, r, sigma, N)
```

Out[2]:
```
1.2857395761264745
```

- Great. Now let's look at the speed:

In [3]:
```
%timeit calltree(S0, K, T, r, sigma, N)  #ipython magic for timing things.
```

```
10 loops, best of 3: 162 ms per loop
```

- Loops tend to be slow in Python. It is often preferable to write code in a *vectorized* style.
- This means calling NumPy `ufuncs` on entire vectors of data, so that the looping happens inside NumPy, i.e., in compiled C code (which means it's fast).

In [4]:
```
def calltree_numpy(S0, K, T, r, sigma, N):
    """
    European call price based on an N-step binomial tree.
    """
    deltaT = T/float(N)
    u = np.exp(sigma * np.sqrt(deltaT))
    d = 1/u
    p = (np.exp(r*deltaT) - d)/(u-d)
    piu = np.exp(-r*deltaT)*p
    pid = np.exp(-r*deltaT)*(1-p)
    C = np.zeros((N+1, N+1))
    S = S0 * u**np.arange(N+1) * d**(2*np.arange(N+1)[:, np.newaxis])
    S = np.triu(S)  #Keep only the upper triangular part.
    C[:, N] = np.maximum(0, S[:, N]-K) #Note: np.maximum in place of max.
    for j in xrange(N-1, -1, -1):
        C[:j+1, j] = piu * C[:j+1, j+1] + pid * C[1:j+2, j+1]
    return  C[0, 0]
```

- Let's verify that both implementations give the same answer.
- We'll use NumPy's `allclose` function, which tests if all elements of two arrays are 'close' to one another (hence avoiding floating point precision issues).

In [5]:
```
np.allclose(calltree(S0, K, T, r, sigma, N), calltree_numpy(S0, K, T, r, sigma, N))
```

Out[5]:
```
True
```

- Now let's time it:

In [6]:
```
%timeit calltree_numpy(S0, K, T, r, sigma, N)
```

```
100 loops, best of 3: 4.39 ms per loop
```


- A third option is to use Numba (user guide).
- Numba implements a *just in time compiler*. It can compile certain (array-heavy) code to native machine code.
- If Numba is able to compile your code, then the speed is often comparable to C.
- All we need to do is import the package, and then add a *decorator* to our function.
- Other than that, the code is exactly the same as our first attempt.

In [7]:
```
from numba import jit
@jit(nopython=True)  #Throw an error if the function cannot be compiled.
def calltree_numba(S0, K, T, r, sigma, N):
    """
    European call price based on an N-step binomial tree.
    """
    deltaT = T/float(N)
    u = np.exp(sigma * np.sqrt(deltaT))
    d = 1/u
    p = (np.exp(r*deltaT) - d)/(u-d)
    C = np.zeros((N+1, N+1))
    S = np.zeros((N+1, N+1))
    piu = np.exp(-r*deltaT)*p
    pid = np.exp(-r*deltaT)*(1-p)
    for i in xrange(N+1):
        for j in xrange(i, N+1):
            S[i, j] = S0 * u**j * d**(2*i)
    for i in xrange(N+1):
        C[i, N] = max(0, S[i, N]-K)
    for j in xrange(N-1, -1, -1):
        for i in xrange(j+1):
            C[i, j] = piu * C[i, j+1] + pid * C[i+1, j+1]
    return  C[0, 0]
```


- Check that it gives the right answer:

In [8]:
```
np.allclose(calltree(S0, K, T, r, sigma, N), calltree_numba(S0, K, T, r, sigma, N))
```

Out[8]:
```
True
```

- The moment of truth:

In [9]:
```
%timeit calltree_numba(S0, K, T, r, sigma, N)
```

```
100 loops, best of 3: 4.94 ms per loop
```

- Not bad at all. We essentially match our NumPy implementation.
- There's one more thing we might try: what if we JIT-compile the vectorized version?
- Instead of writing out the whole function again, we'll use an alternative way to invoke the JIT compiler:

```
In [10]: calltree_numpy_numba=jit(calltree_numpy)
         np.allclose(calltree(S0, K, T, r, sigma, N), calltree_numpy_numba(S0, K, T, r, sigma, N))
Out[10]: True
In [11]: %timeit calltree_numpy_numba(S0, K, T, r, sigma, N)
         1000 loops, best of 3: 1.67 ms per loop
```

- Wow, can't hate that.
- Looking at the absolute timings, the improvements may seem small, but keep in mind that you may need to call these functions many many times.
- Other tools for compiling Python to native code include Cython and Pythran.

A Closed Form for European Options

- The price of a European option
$$C_0 = e^{-rT}\mathbb{E}^{\mathbb{Q}}\left[\max(S_T - K), 0\right]$$
depends only on S_T, so there is no need to use a tree explicitly to evaluate it.
- Let k denote the number of up moves of the stock , so that $N - k$ is the number of down moves. Then
$$S_T = S_0 u^k d^{N-k} = S_0 u^{2k-N},$$
where under $\mathbb{Q}$, $k \sim \text{Bin}(N, p)$, with pmf $f(k; N, p) = \binom{N}{k} p^k (1-p)^{N-k}$. Thus
$$C_0 = e^{-rT} \sum_{k=0}^{N} f(k; N, p) \max(S_0 u^k d^{N-k} - K, 0).$$

- Let a denote the minimum number of up moves so that $S_T > K$, i.e., the smallest integer greater than $N/2 + \log(K/S_0)/(2\log u)$. Then
$$C_0 = e^{-rT} \sum_{k=a}^{N} f(k; N, p) \left[S_0 u^k d^{N-k} - K\right].$$
- The second term is $[1 - F(a-1; N, p)]e^{-rT}K = \bar{F}(a-1; N, p)e^{-rT}K$, where F is the binomial cdf and $\bar{F}$ is the survivor function.
- Let $p_* = e^{-r\delta t}pu$. The first term is
$$e^{-rT}S_0 \sum_{k=a}^{N} \binom{N}{k} p^k (1-p)^{N-k} u^k d^{N-k} = S_0 \sum_{k=a}^{N} \binom{N}{k} p_*^k (1-p_*)^{N-k}.$$

- Putting things together,
$$\begin{aligned} C_0 &= S_0 \bar{F}(a-1; N, p_*) - \bar{F}(a-1; N, p)e^{-rT}K \\ &= S_0 \mathbb{Q}^*(S_T > K) - \mathbb{Q}(S_T > K)e^{-rT}K \end{aligned}$$
- You will be implementing this in a homework exercise.

The Black-Scholes Formula as Continuous Time Limit

- Let's consider what happens if we let $N \to \infty$
- A first-order Taylor expansion, together with l'Hopital's rule, can be used to show that, for small δt,

$$p \approx \frac{1}{2}\left(1 + \sqrt{\delta t}\frac{r - \frac{1}{2}\sigma^2}{\sigma}\right).$$

- Similarly,

$$p^* \approx \frac{1}{2}\left(1 + \sqrt{\delta t}\frac{r + \frac{1}{2}\sigma^2}{\sigma}\right).$$

- Next, Let $X_T \equiv \log S_T$. Then, because R_i is either $\log u$ or $\log d = -\log u$,

$$X_T = \log S_0 + \sum_{i=1}^{N} R_i = \log S_0 + \sigma\sqrt{\delta t}(2k - N).$$

- As $k \sim \text{Bin}(N, p)$, we have $\mathbb{E}^{\mathbb{Q}}[k] = Np$ and $\text{var}^{\mathbb{Q}}[k] = Np(1-p)$.
- Thus,

$$\mathbb{E}^{\mathbb{Q}}[X_T] = \log S_0 + \sigma\sqrt{\delta t}N(2p-1) \to \log S_0 + (r - \frac{1}{2}\sigma^2)T$$
$$\text{Var}^{\mathbb{Q}}[X_T] = \sigma^2 \delta t 4Np(1-p) \to \sigma^2 T.$$

- Finally, as $N \to \infty$, the distribution of X_T tends to a normal. This follows from the *central limit theorem* and the fact that X_T is the sum of N i.i.d. terms.

- Thus, as $N \to \infty$,

$$\mathbb{Q}(S_T > K) = \mathbb{Q}(X_T > \log K) = \mathbb{Q}\left(\frac{X_T - \mathbb{E}^{\mathbb{Q}}[X_T]}{\sqrt{\text{Var}^{\mathbb{Q}}[X_T]}} > \frac{\log K - \mathbb{E}^{\mathbb{Q}}[X_T]}{\sqrt{\text{Var}^{\mathbb{Q}}[X_T]}}\right)$$
$$= 1 - \Phi\left(\frac{\log K - \mathbb{E}^{\mathbb{Q}}[X_T]}{\sqrt{\text{Var}^{\mathbb{Q}}[X_T]}}\right) =: 1 - \Phi(-d_2) = \Phi(d_2), \text{ where}$$
$$d_2 \equiv \frac{\mathbb{E}^{\mathbb{Q}}[X_T] - \log K}{\sqrt{\text{Var}^{\mathbb{Q}}[X_T]}} = \frac{\log(S_0/K) + (r - \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}.$$

- The same argument can be used to show that as $N \to \infty$, $\mathbb{Q}^*(S_T > K) = \Phi(d_1)$, where

$$d_1 \equiv d_2 + \sigma\sqrt{T} = \frac{\log(S_0/K) + (r + \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}.$$

- In summary, we have derived the *Black-Scholes formula*

$$C_0 = S_0\Phi(d_1) - e^{-rT}K\Phi(d_2)$$
$$=: BS(S_0, K, T, r, \sigma).$$

- Implementation in Python:

In [12]:
```
from scipy.stats import norm
def blackscholes(S0, K, T, r, sigma):
    """
    Price of a European call in the Black-Scholes model.
    """
    d1 = (np.log(S0)-np.log(K)+(r+sigma**2/2)*T)/(sigma*np.sqrt(T))
    d2 = d1-sigma*np.sqrt(T)
    return S0*norm.cdf(d1)-np.exp(-r*T)*K*norm.cdf(d2)
```

In [13]:
```
calltree(S0, K, T, r, sigma, 500), blackscholes(S0, K, T, r, sigma)
```

Out[13]:
```
(1.2857395761264745, 1.2858368491569285)
```

- Note that as written, the function can operate on arrays of strikes:

In [14]:
```
Ks = np.linspace(8, 10, 5)
blackscholes(S0, Ks, T, r, sigma)
```

Out[14]:
```
array([ 3.04764278,  2.56561793,  2.10292676,  1.67202011,  1.28583685])
```


American Options

- Unlike a European call, an American call with price C_t^{Am} can be exercised at any time before it matures. When exercised at $t \leq T$, it pays $\max(S_t - K, 0)$. Hence the call will be exercised early if at time t, $S_t - K > C_t^{Am}$.
- Recall put-call parity: $C_t - P_t = S_t - e^{-r(T-t)}K$, which implies (for r>0)

$$C_t \geq S_t - e^{-r(T-t)}K \geq S_t - K,$$
$$P_t \geq Ke^{-r(T-t)} - S_t.$$

- As $C_t^{Am} \geq C_t$, an American call is therefore never exercised early (in the absence of dividends).
- There is no closed-form expression for the price of an American put option, so numerical methods are needed. Binomial trees are a popular choice.

- This works as follows:
  - At step N, the price of the put is $P_N^{Am} = \max(K - S_N, 0)$, just like for a European put.
  - At step $N-1$, the *continuation value* of the option is $e^{-r\delta t}\mathbb{E}^{\mathbb{Q}}[P_N^{Am}]$. Early exercise yields $K - S_{N-1}$, so
$$P_{N-1}^{Am} = \max(e^{-r\delta t}\mathbb{E}^{\mathbb{Q}}[P_N^{Am}|\mathcal{F}_{N-1}], K - S_{N-1}).$$
  - This is iterated backwards until P_0^{Am}.
- The implementation is part of the homework exercise.

Implied Volatility

- The *implied volatility* (IV, σ_I) of an option is that value of σ which equates the BS model price to the observed market price C_0^{obs}, i.e., it solves
$$C_0^{obs} = BS(S_0, K, T, r, \sigma_I).$$
- If the BS assumptions were correct, then any option traded on the asset should have the same IV, which should in turn equal historical volatility.
- In practice, options with different strikes K and hence *moneyness* K/S_0 have different IVs: *volatility smile* or *smirk/skew*. Also, options with different times to maturity have different IVs: *volatility term structure*.
- These phenomena are evidence of a failure of the assumptions of the Black-Scholes model, most importantly that of a constant volatility σ.

- In practice, the BS formula is used as follows: the implied volatility is computed for options that are already traded in the market, for different strikes and maturities. This leads to the *IV surface*.
- When a new option is issued, the implied volatility corresponding to its strike and time to maturity is determined by interpolation on the surface. The BS formula then gives the corresponding price.
- Mathematically, the IV is the *root* (or *zero*) of the function

$$f(\sigma_I) = BS(S_0, K, T, r, \sigma_I) - C_0^{obs}.$$

- In Python, root finding can be done via SciPy's `brentq` function. In its simplest form, it takes 3 arguments: the unary function $f(\cdot)$, and a lower bound L and upper bound U such that $[L, U]$ contains exactly one root of f.

- Tehranchi (2016) shows that for European calls,

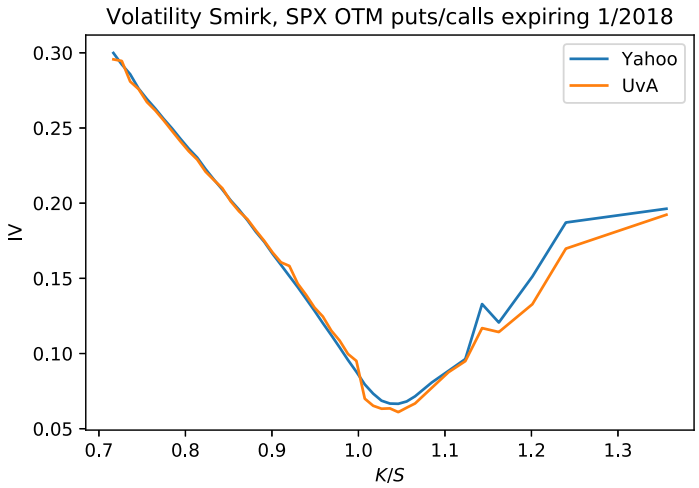
$$-\Phi^{-1}\left(\frac{S_0 - C_0^{obs}}{2\min(S_0, e^{-rT}K)}\right) \leq \frac{\sqrt{T}}{2}\sigma_I \leq -\Phi^{-1}\left(\frac{S_0 - C_0^{obs}}{S_0 + e^{-rT}K}\right).$$

- It remains to transform our objective function into a unary (single argument) function, through *partial function application* via, e.g., an anonymous function:

In [15]:
```
from scipy.optimize import brentq
def impvol(S0, K, T, r, C_obs, Type='call'):
    """Implied Black-Scholes volatility."""
    if Type == 'put':  #Convert to call price via parity
        C_obs = C_obs+S0-np.exp(-r*T)*K
    L = -2*norm.ppf((S0-C_obs)/(2.0*min(S0, np.exp(-r*T)*K)))/np.sqrt(T)
    U = -2*norm.ppf((S0-C_obs)/(S0+np.exp(-r*T)*K))/np.sqrt(T)
    return brentq(lambda s: blackscholes(S0, K, T, r, s)-C_obs, L, U)  #Partial application: f(s)=BS(S0, K, T,
```

In [16]:
```
C_obs=2.  #For illustration.
IV=impvol(S0, K, T, r, C_obs); (IV, blackscholes(S0, K, T, r, IV))
```

Out[16]:
```
(0.6802980451017724, 1.9999999999991251)
```

Computational Finance

AMSTERDAM BUSINESS SCHOOL **Business**

Monte Carlo Methods

Brownian Motion

- We saw last week that the binomial tree implies for $X_t \equiv \log S_t$ that
$$X_{i\delta t} = X_{(i-1)\delta t} + R_i \iff \delta X_t = R_i, \qquad (\dagger)$$
where $R_i = \log u$ or $R_i = \log d$, with probabilities $\mathbb{Q}[u]$ and $\mathbb{Q}[d]$.
- Equation (†) is a *stochastic difference equation.*
- Its *solution*
$$X_T = \log S_0 + \sum_{i=1}^{N} R_i = \log S_0 + \sigma\sqrt{\delta t}(2k - N)$$
is called a *binomial process*, or in the special case with $\mathbb{E}[R_i] = 0$, a *random walk*.

- We also saw that if we let $N \to \infty$ (so that $\delta t \to 0$),
$$X_T - X_0 \xrightarrow{d} N(\mu T, \sigma^2 T), \qquad \mu \equiv r - \frac{1}{2}\sigma^2.$$
- The argument can be repeated for every X_t, $t \leq T$, showing that
$$X_t - X_0 \xrightarrow{d} N(\mu t, \sigma^2 t),$$
and that for any $0 < t < T$, $X_t - X_0$ and $X_T - X_t$ are independent.
- As $\delta t \to 0$, $\{X_t\}_{t\geq 0}$ becomes a continuous time process: the indexing set is now given by the entire positive real line.
- This continuous time limit (with $\mu = 0$ and $\sigma^2 = 1$) is called *Brownian motion*, or *Wiener process.*
- From now on, rather than modelling in discrete time and then letting $\delta t \to 0$, we will directly model in continuous time, using Brownian motion as a building block.

- Definition of (standard) *Brownian Motion*: Stochastic process $\{W_t\}_{t\geq 0}$ satisfying
  - $W_0 = 0$;
  - The increments $W_t - W_s$ are independent for all $0 \leq s < t$;
  - $W_t - W_s \sim N(0, t-s)$ for all $0 \leq s \leq t$;
  - Continuous sample paths.

- Properties of Brownian Sample Paths:
  - *Continuity*: by assumption, and also $W_{t+\delta t} - W_t \sim N(0, \delta t) \to 0$ as $\delta t \downarrow 0$;
  - *Nowhere differentiability*: intuitively, this is seen from
    $$\frac{W_t - W_{t-\delta t}}{\delta t} \sim N\left(0, \frac{1}{\delta t}\right), \quad \frac{W_{t+\delta t} - W_t}{\delta t} \sim N\left(0, \frac{1}{\delta t}\right);$$
    left and right difference quotients do not have (common) limit as $\delta t \downarrow 0$.
  - *Self-similarity*: Zooming in on a Brownian motion yields another Brownian motion: for any $c > 0, X_t = \sqrt{c} W_{t/c}$ is a Brownian motion.

Simulating Brownian Motion

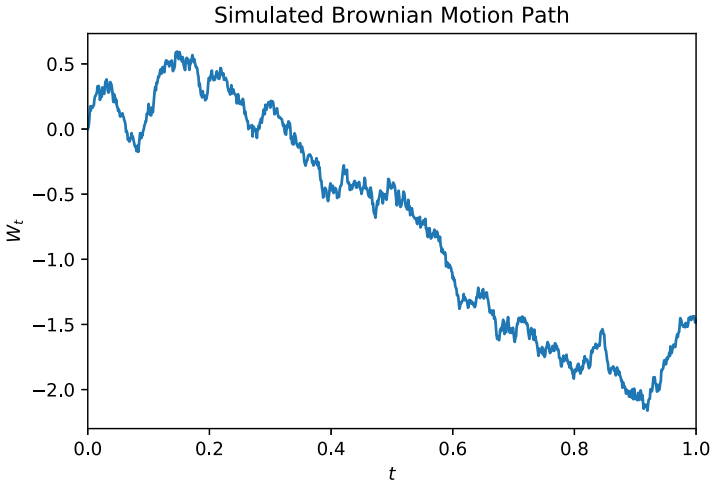
- In order to simulate Brownian Motion, we need to discretize it. As usual, split the time interval $[0, T]$ into N parts of length $\delta t = T/N$ and let $t_i \equiv i\delta t$.
- As in (†), let
  $$W_{i\delta t} = W_{(i-1)\delta t} + R_i \iff \delta W_t = R_i,$$
  but where now we model R_i as normal rather than binomial:
  $$R_i = \sqrt{\delta t} \cdot Z_i, \quad Z_i \sim \mathrm{N}(0, 1).$$
- As a slight generalization, *Brownian motion with drift* is obtained from
  $$\delta X_t \equiv X_{i\delta t} - X_{(i-1)\delta t} = \mu \delta t + \sigma \delta W_t = \mu \delta t + \sigma \sqrt{\delta t} Z_i,$$
  and where we allow X_0 to be an arbitrary value.
- This implies
  $$X_t \equiv X_0 + \mu t + \sigma W_t,$$
  so that $\mathbb{E}[X_t] = X_0 + \mu t, \mathrm{Var}[X_t] = \sigma^2 t$. Hence the average upwards (or downwards if $\mu < 0$) tendency over a time interval δt is $\mu \delta t$.

- In order to implement this, we need a way of drawing random samples from the normal distribution.
- Computers are deterministic machines. They cannot generate true random numbers.
- Instead, they construct sequences of pseudo-random numbers from a specified distribution that *look* random, in the sense that they pass certain statistical tests.
- E.g., NumPy's `np.random.randn(d0[, d1, ...])` constructs an array of standard normal pseudo random numbers.
- Random number generators use a *seed* value for initialization. Given the same seed, the same pseudo-random sequence will be returned.
- NumPy picks the seed automatically. To force it to use a specific seed, use `np.random.seed(n)`. Putting this line at the beginning of your Monte-Carlo program ensures that you get exactly the same results every time the program is run.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
%matplotlib inline
```

In [2]:
```
def bmsim(T, N, X0=0, mu=0, sigma=1):
    """Simulate a Brownian motion path.
    """
    deltaT = float(T)/N
    tvec = np.linspace(0, T, N+1)
    z = np.random.randn(N+1)  #N+1 is one more than we need, actually. This way we won't have to grow dX by X0
    dX = mu*deltaT + sigma*np.sqrt(deltaT)*z  #X[j+1]-X[j]=mu*deltaT + sigma*np.sqrt(deltaT)*z[j].
    dX[0] = 0.
    X = np.cumsum(dX)
    X += X0
    return tvec, X
```

In [3]:
```
np.random.seed(0)
tvec, W = bmsim(1, 1000)
W = pd.Series(W, index=tvec)
W.plot()
plt.title('Simulated Brownian Motion Path')
plt.xlabel("$t$"); plt.ylabel("$W_t$");
plt.savefig("img/BMpath.svg"); plt.close()
```



Ito Processes

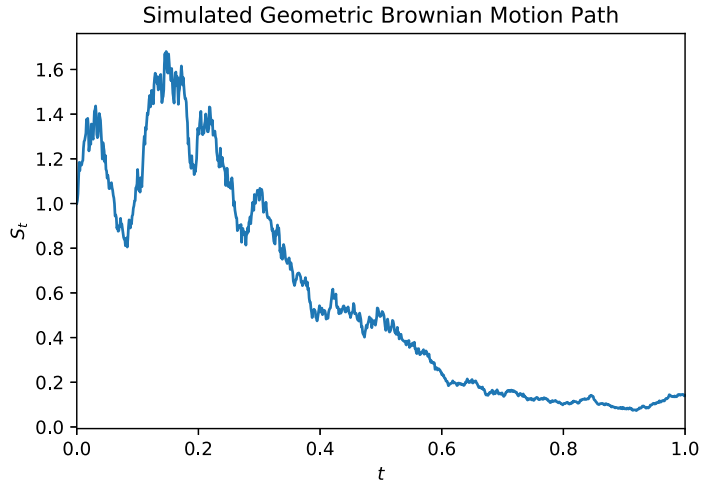
- Ito processes generalize Brownian motion with drift by allowing the drift and volatility to be time-varying:
$$\delta X_t \equiv X_{i\delta t} - X_{(i-1)\delta t} = \mu_{t_{i-1}}\delta t + \sigma_{t_{i-1}}\delta W_t.$$
- We allow μ_{t_i} and σ_{t_i} to be stochastic; e.g., they may depend on X_{i-1}, as in
$$\delta X_t = \mu(t_{i-1}, X_{i-1})\delta t + \sigma(t_{i-1}, X_{i-1})\delta W_t.$$
- One can show that under appropriate conditions, the process has a well-defined limit as $N \to \infty$ (so that $\delta t \downarrow 0$), for which we write
$$dX_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t.$$
This is known as a stochastic differential equation (SDE).
- If σ_t is stochastic, then the distribution of X_T implied by the SDE is no longer normal. One can show that if N is large enough, then this distribution is correctly reproduced by the discretized version (which is then known as as the *Euler approximation*).

- Remark: in continuous time, a process $\{X_t\}_{t\geq 0}$ is a *martingale* if
  - $\mathbb{E}[|X_t|] < \infty$, for all $t \geq 0$;
  - $\mathbb{E}[X_t|\mathcal{F}_s] = X_s$, for all $t > s \geq 0$, where $\mathcal{F}_t$ denotes the information on X_t up to time t. An Ito process is a martingale if and only if the drift is zero, because (using $s = t_{i-1}$)
$$\begin{aligned}\mathbb{E}[\delta X_t|\mathcal{F}_{t_{i-1}}] &= \mathbb{E}[\mu(t_{i-1}, X_{i-1})\delta t + \sigma(t_{i-1}, X_{i-1})\delta W_t|\mathcal{F}_{t_{i-1}}] \\ &= \mu(t_{i-1}, X_{i-1})\delta t + \sigma(t_{i-1}, X_{i-1})\mathbb{E}[\delta W_t|\mathcal{F}_{t_{i-1}}] \\ &= \mu(t_{i-1}, X_{i-1})\delta t + \sigma(t_{i-1}, X_{i-1})\mathbb{E}[W_{t_i} - W_{t_{i-1}}|\mathcal{F}_{t_{i-1}}] \\ &= \mu(t_{i-1}, X_{i-1})\delta t.\end{aligned}$$

- **Example**: Take $\mu(t, S_t) \equiv \mu S_t$ and $\sigma(t, S_t) \equiv \sigma S_t$. The resulting process
$$dS_t = \mu S_t dt + \sigma S_t dW_t$$
is known as *Geometric Brownian Motion*. Its Euler approximation is
$$\delta S_t \equiv S_i - S_{i-1} = \mu S_{i-1}\delta t + \sigma S_{i-1}\sqrt{\delta t}Z_i.$$

In [4]:
```
def gbmsim(T, N, S0=1, mu=0, sigma=1):
    """Simulate a Geometric Brownian motion path.
    """
    deltaT = float(T)/N
    tvec = np.linspace(0, T, N+1)
    z = np.random.randn(N+1)  #Again one more than we need. This keeps it comparable to bmsim.
    S = np.zeros_like(z)
    S[0] = S0
    for i in xrange(0, N): #Note: we can no longer vectorize this, because S[:, j] is needed for S[:, j+1].
        S[i+1] = S[i] + mu*S[i]*deltaT + sigma*S[i]*np.sqrt(deltaT)*z[i+1]
    return tvec, S
```

In [5]:
```
np.random.seed(0)
tvec, S = gbmsim(1, 1000)
S = pd.Series(S, index=tvec)
S.plot()
plt.title('Simulated Geometric Brownian Motion Path')
plt.xlabel("$t$"); plt.ylabel("$S_t$")
plt.savefig("img/GBMpath.svg"); plt.close()
```



Ito's Lemma

- Ito's lemma answers the question: if X_t is an Ito process with given dynamics (SDE), then what are the dynamics of a function $f(t, X_t)$?
- It can be stated as follows: Let $\{X_t\}_{t\geq 0}$ be an Ito process satisfying $dX_t = \mu_t dt + \sigma_t dW_t$, and consider a function $f : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R}$ with continuous partial derivatives

$$\dot{f}(t,x) = \frac{\partial f(t,x)}{\partial t}, \qquad f'(t,x) = \frac{\partial f(t,x)}{\partial x}, \qquad f''(t,x) = \frac{\partial^2 f(t,x)}{\partial x^2}.$$

Then

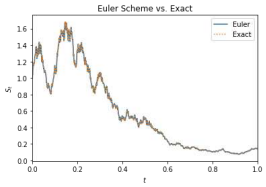
$$df(t,X_t) = \dot{f}(t,X_t)dt + f'(t,X_t)dX_t + \frac{1}{2}f''(t,X_t)\sigma_t^2 dt.$$

- **Example**: Geometric Brownian Motion. Let $dS_t = \mu S_t dt + \sigma S_t dW_t$ and $X_t = f(S_t) = \log S_t$. Then $\dot{f}(S_t) = 0, f'(S_t) = 1/S_t, f''(S_t) = -1/S_t^2$, and

$$\begin{aligned} dX_t = df(S_t) &= \dot{f}(S_t)dt + f'(S_t)dS_t + \frac{1}{2}f''(S_t)(S_t\sigma)^2 dt \\ &= \frac{1}{S_t}dS_t - \frac{1}{2S_t^2}(S_t\sigma)^2 dt \\ &= \frac{1}{S_t}(\mu S_t dt + \sigma S_t dW_t) - \frac{1}{2}\sigma^2 dt \\ &= \nu dt + \sigma dW_t, \quad \nu = \mu - \frac{1}{2}\sigma^2. \end{aligned}$$

- So we find that $S_t = \exp(X_t) = S_0 \exp(\nu t + \sigma W_t)$. This gives us an alternative way to simulate GBM, which avoids the error introduced by the Euler approximation.
- We see that S_T has a lognormal distribution in the GBM model, and that S_t is a martingale (driftless process) if and only if $\mu = 0$, so that $\nu = -\frac{1}{2}\sigma^2$.

In [6]:
```
N=1000  #Try changing N to 100, then 10!
np.random.seed(0)
tvec, S1 = gbmsim(1, N)  #mu = 0, sigma = 1
np.random.seed(0)  #Use the same seed, otherwise we'd get different paths.
tvec, X = bmsim(1, N, 0, -.5)  #nu = mu - .5*sigma**2 = -.5
S2 = np.exp(X)
S1 = pd.Series(S1, index=tvec)
S2 = pd.Series(S2, index=tvec)
S1.plot()
S2.plot(linestyle=":")
plt.title("Euler Scheme vs. Exact")
plt.xlabel("$t$")
plt.ylabel("$S_t$")
plt.legend(["Euler", "Exact"]);
```



- Intuition for Ito's lemma: (see Hull, 2012, Appendix to Ch. 13): In standard calculus, the total differential
$$df = \dot{f}(t, g(t))dt + f'(t, g(t))dg(t)$$
is the linear part of a Taylor expansion; the remaining terms are of smaller order as $dt, dg(t) \to 0$, so the total differential is a local linear approximation to f.
- If $g(t) = X_t$, an Ito process, take a 2nd order Taylor approximation:
$$\delta f \approx \dot{f}(t, X_t)\delta t + f'(t, X_t)\delta X_t + \frac{1}{2}\left[\frac{\partial^2 f}{\partial t^2}(\delta t)^2 + 2\frac{\partial^2 f}{\partial t \partial X_t}(\delta t)(\delta X_t) + \frac{\partial^2 f}{\partial X_t^2}(\delta X_t)^2\right].$$

- We have that $\delta X_t \equiv (X_t - X_{t-\delta t}) \approx \mu_{t-\delta t}\delta t + \sigma_{t-\delta t}\delta W_t \sim N(\mu_{t-\delta t}\delta t, \sigma^2_{t-\delta t}\delta t)$. Thus, $\mathbb{E}[(\delta X_t)^2] \approx (\mu_{t-\delta t}\delta t)^2 + \sigma^2_{t-\delta t}\delta t \approx \sigma^2_{t-\delta t}\delta t$; i.e., the 2nd order term is of the same order of magnitude as the 1st order term δt.
- It can be shown that as $\delta t \to 0$, $(\delta X_t)^2$ can be treated as non-stochastic: $(dX_t)^2 = \sigma_t^2 dt$. Together with $(dt)^2 = 0$ and $(dt)(dX_t) = 0$, this gives the result.

The Black-Scholes Model

- Black and Scholes assumed the following model:
  - The stock $\{S_t\}_{t\in[0,T]}$ follows GBM:
$$dS_t = \mu S_t dt + \sigma S_t dW_t.$$
  - The stock pays no dividends.
  - Cash bond price $B_t = e^{rt} \iff dB_t = rB_t dt$; riskless lending and borrowing at the same rate r.
  - European style derivative with price C_t and payoff $C_T = (S_T)$.
  - Trading may occur continuously, with no transaction costs.
  - No arbitrage opportunities.
- The problem is to find the price $C_t, t \in [0, T]$.

- It can be shown that the FTAP holds in continuous time as well: if the market is arbitrage free, then there exists a risk neutral measure $\mathbb{Q}$ under which all assets earn the risk free rate (on average), and the price of a claim is the discounted expected payoff under $\mathbb{Q}$. If the market is complete, then $\mathbb{Q}$ is unique. This gives us a pricing formula for general European claims:
$$C_t = e^{-r(T-t)}\mathbb{E}^{\mathbb{Q}}\left[C_T | \mathcal{F}_t\right].$$
- This implies that if we can simulate the stock price under the measure $\mathbb{Q}$, then we can price the claim by Monte Carlo simulation.

- In the BS model, it can be shown that under the risk-neutral measure $\mathbb{Q}$,
$$dS_t = rS_t dt + \sigma S_t dW_t.$$
- Note that by Ito's lemma, the discounted stock price $\tilde{S}_t \equiv e^{-rt}S_t =: f(t, S_t)$ satisfies
$$\begin{aligned} d\tilde{S}_t &= \dot{f}(t, S_t)dt + f'(t, S_t)dS_t + \frac{1}{2}f''(t, S_t)\sigma^2 S_t^2 dt \\ &= -re^{-rt}S_t dt + e^{-rt}dS_t + 0 \\ &= -r\tilde{S}_t dt + e^{-rt}(rS_t dt + \sigma S_t dW_t) \\ &= \sigma\tilde{S}_t dW_t. \end{aligned}$$
- I.e., $\tilde{S}_t$ is an Ito process without drift, and thus a martingale. This is the reason that $\mathbb{Q}$ is also called the equivalent martingale measure.

- We can extend the BS model by assuming that the underlying pays a continuous dividend at rate δ (realistic only for indices, not individual stocks). Then a position of 1 share generates an instantaneous dividend stream $\delta S_t dt$, in addition to the capital gains dS_t. Note that only the holder of the underlying receives the dividend.
- The pricing formula remains the same, but now the risk-neutral dynamics of S_t are
$$dS_t = (r - \delta)S_t dt + \sigma S_t dW_t.$$
- The expected growth rate of the underlying under $\mathbb{Q}$ is $r - \delta$, so the expected return from holding it (capital gains plus dividend yield) is r.
- The price of a call is now
$$C_t = e^{-\delta(T-t)}S_t\Phi(d_1) - e^{-r(T-t)}K\Phi(d_2),$$
where
$$d_{1,2} = \frac{\log(S_t/K) + [(r - \delta) \pm \frac{1}{2}\sigma^2](T - t)}{\sigma\sqrt{T - t}}.$$

Monte Carlo Pricing

- The goal in Monte Carlo simulation is to obtain an estimate of
$$\theta \equiv \mathbb{E}[X],$$
for some random variable X with finite expectation. The assumption is that we have a means of sampling from the distribution of X, but no closed-form expression for θ.
- Suppose we have a sample $\{X_i\}_{i\in\{1,\dots,n\}}$ of *independent* draws for X, and let
$$\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i.$$
- The sample average $\bar{X}_n$ is an *unbiased estimator* of θ: $\mathbb{E}[\bar{X}_n] = \theta$.
- The *weak law of large numbers* states that
$$\bar{X}_n \xrightarrow{p} \theta,$$
where the arrow denotes *convergence in probability*; i.e., as the sample size grows, $\bar{X}_n$ becomes a better and better estimate of θ.

- Thus, our strategy is to use a computer to draw n (pseudo) random numbers X_i from the distribution of X, and then estimate θ as the sample mean of the X_i.
- n is called the number of *replications*.
- For finite n, the sample average will be an approximation to θ.
- It is usually desirable to have an estimate of the accuracy of this approximation. Such an estimate can be obtained from the *central limit theorem* (CLT), which states that
$$\sqrt{n}(\bar{X}_n - \theta) \xrightarrow{d} N\left(0, \sigma^2\right),$$
provided that σ^2, the variance of X, is finite. The arrow denotes convergence in distribution; this implies that for large n, $\bar{X}_n$ has approximately a normal distribution.
- Of course σ^2 is unknown, but we can estimate it as
$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}\left(\bar{X}_n - X_i\right)^2.$$

- A 95% *confidence interval* (CI) is an interval $[c_l, c_u]$ with $\mathbb{P}[c_l \leq \theta \leq c_u] = 0.95$.
- The CLT implies that, in the limit as $n \to \infty$,

$$\mathbb{P}[-1.96\sigma \leq \sqrt{n}(\bar{X}_n - \theta) \leq 1.96\sigma] = 0.95 \Leftrightarrow$$
$$\mathbb{P}[\bar{X}_n - 1.96\frac{\sigma}{\sqrt{n}} \leq \theta \leq \bar{X}_n + 1.96\frac{\sigma}{\sqrt{n}}] = 0.95.$$

- Hence $c_l = \bar{X}_n - 1.96\frac{\sigma}{\sqrt{n}}$ and $c_u = \bar{X}_n + 1.96\frac{\sigma}{\sqrt{n}}$ is an asymptotically valid CI.
- Note that c_l and c_u are random variables; we should interpret this as "before the experiment is performed, there is a 95% chance that a CI computed according to this formula will contain θ". After performing the experiment, this statement is not valid anymore; the interval is now fixed, and contains θ with probability either 0 or 1.
- The unknown parameter σ can be consistently estimated by $\sqrt{\hat{\sigma}^2}$.

Application: Asian Options

- The payoff of Asian options depends on the *average* price of the underlying, $\bar{S}_T$. Types:
  - Average price Asian call with payoff $(\bar{S}_T - K)^+$;
  - Average price Asian put with payoff $(K - \bar{S}_T)^+$;
  - Average strike Asian call with payoff $(S_T - \bar{S}_T)^+$;
  - Average strike Asian put with payoff $(\bar{S}_T - S_T)^+$.
- It is important to specify how the average is computed: the continuous, arithmetic, and geometric averages are, respectively,

$$\frac{1}{T}\int_0^T S_t dt, \qquad \frac{1}{N}\sum_{i=1}^N S_{t_i}, \qquad \text{and} \qquad \left(\prod_{i=1}^N S_{t_i}\right)^{1/N},$$

where the t_i are a set of N specified dates.

- Exact Black-Scholes type pricing formulas for Asian options only exist in special cases (e.g., the geometric average Asian call, see next week), so we rely on Monte Carlo simulation.
- Our pricing formula

$$C_t = e^{-r(T-t)}\mathbb{E}^{\mathbb{Q}}[C_T|\mathcal{F}_t]$$

is exactly in the required form.
- As an example, consider pricing an arithmetic average price call with payoff

$$C_T = (\bar{S}_T - K)^+, \quad \text{where} \quad \bar{S}_T = \frac{1}{N}\sum_{i=1}^N S_{t_i},$$

which cannot be priced analytically.
- The payoff is path-dependent, so we need to simulate the entire asset price path, not just S_T.

In [7]:
```
from scipy.stats import norm
def asianmc(S0, K, T, r, sigma, delta, N, numsim=10000):
    """Monte Carlo price of an arithmetic average Asian call.
    """
    X0 = np.log(S0)
    nu = r-delta-.5*sigma**2
    payoffs = np.zeros(numsim)
    for i in xrange(numsim):
        _, X = bmsim(T, N, X0, nu, sigma)  #Convention: underscore holds value to be discarded.
        S = np.exp(X)
        payoffs[i] = max(S[1:].mean()-K, 0.)
    g = np.exp(-r*T)*payoffs
    C = g.mean(); s = g.std()
    zq = norm.ppf(0.975)
    Cl = C-zq/np.sqrt(numsim)*s
    Cu = C+zq/np.sqrt(numsim)*s
    return C, Cl, Cu
```

In [8]:
```
S0 = 11; K = 10; T = 3/12.; r = 0.02; sigma = .3; delta = 0.01; N = 10
np.random.seed(0)
C0, Cl, Cu = asianmc(S0, K, T, r, sigma, delta, N); C0, Cl, Cu
```

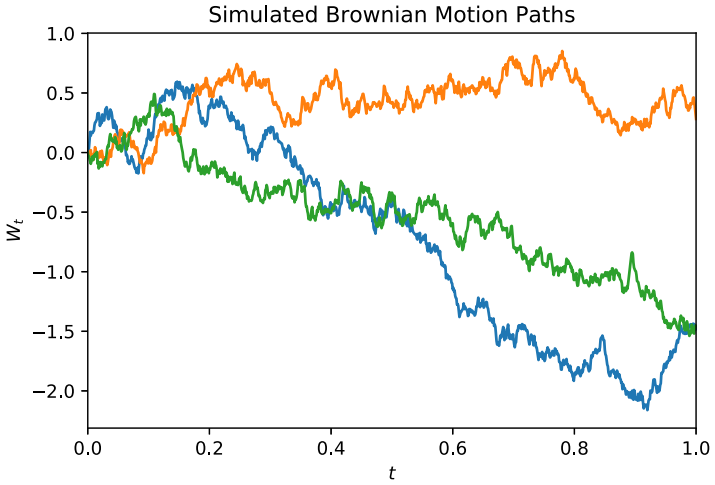
Out[8]: (1.0927262054551385, 1.0747653929130998, 1.1106870179971773)

Code Optimization

- Our code for pricing the Asian option is likely inefficient, because it contains a loop.
- The code can be 'vectorized' to speed it up.
- First step: simulate a bunch of Brownian paths in one shot.
- The resulting code is actually almost identical:

In [9]:
```
def bmsim_vec(T, N, X0=0, mu=0, sigma=1, numsim=1):  #Note new input: numsim, the number of paths.
    """Simulate `numsim` Brownian motion paths.
    """
    deltaT = float(T)/N
    tvec = np.linspace(0, T, N+1)
    z = np.random.randn(numsim, N+1)  #(N+1)->(numsim, N+1)
    dX = mu*deltaT + sigma*np.sqrt(deltaT)*z
    dX[:, 0] = 0.  #dX[0]->dX[:, 0]
    X = np.cumsum(dX, axis=1)  #cumsum(dX)->cumsum(dX, axis=1)
    X += X0
    return tvec, X
```

In [10]:
```
np.random.seed(0)
tvec, W = bmsim_vec(1, 1000, numsim=3)
W = pd.DataFrame(W.transpose(), index=tvec)
W.plot().legend().remove()
plt.title('Simulated Brownian Motion Paths')
plt.xlabel("$t$"); plt.ylabel("$W_t$");
plt.savefig("img/BMpaths.svg"); plt.close()
```

- Here is the vectorized code for the Asian option:

In [11]:
```
def asianmc_vec(S0, K, T, r, sigma, delta, N, numsim=10000):
    """Monte Carlo price of an arithmetic average Asian call.
    """
    X0 = np.log(S0)
    nu = r-delta-.5*sigma**2
    #simulate all paths at once:
    _, X = bmsim_vec(T, N, X0, nu, sigma, numsim)
    S = np.exp(X)
    payoffs = np.maximum(S[:, 1:].mean(axis=1)-K, 0.)  #S[1:]->S[:, 1:], max->maximum, mean()->mean(axis=1)
    g = np.exp(-r*T)*payoffs
    C = g.mean(); s = g.std()
    zq = norm.ppf(0.975)
    Cl = C - zq/np.sqrt(numsim)*s
    Cu = C + zq/np.sqrt(numsim)*s
    return C, Cl, Cu
```

- Let's see if it works:

In [12]:
```
np.random.seed(0)
C0_vec, _, _ = asianmc_vec(S0, K, T, r, sigma, delta, N)
np.allclose(C0_vec, C0)
```

Out[12]:
```
True
```

- And time it:

In [13]:
```
%timeit asianmc(S0, K, T, r, sigma, delta, N)
```

```
1 loop, best of 3: 376 ms per loop
```

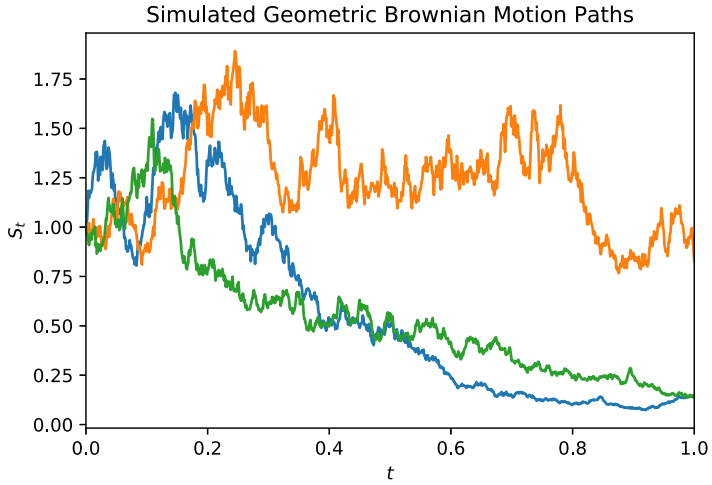
In [14]:
```
%timeit asianmc_vec(S0, K, T, r, sigma, delta, N)
```

```
100 loops, best of 3: 5.93 ms per loop
```


- Our code for the Euler scheme can likewise be adjusted to compute many paths in one shot.
- We're still stuck with the loop over t though, which cannot be vectorized because S_{i+1} depends on S_i.
- We'll use Numba's JIT compiler to speed it up further.

In [15]:
```
from numba import jit
@jit(nopython=True)
def gbmsim_vec(T, N, S0=1, mu=0, sigma=1, numsim=1, seed=0):
    """Simulate `numsim` Geometric Brownian motion paths.
    """
    deltaT = float(T)/N
    tvec = np.linspace(0, T, N+1)
    np.random.seed(seed)  #Note: with jit-compiled functions, the RNG must be seeded INSIDE the compiled code.
    z = np.random.randn(numsim, N+1)
    S = np.zeros_like(z)
    S[:, 0] = S0
    for i in xrange(0, N):
        S[:, i+1]=S[:, i] + mu*S[:, i]*deltaT + sigma*S[:, i]*np.sqrt(deltaT)*z[:, i+1]
    return tvec, S
```

In [16]:
```
tvec, S = gbmsim_vec(1, 1000, numsim=3, seed=0)
S = pd.DataFrame(S.transpose(), index=tvec)
S.plot().legend().remove()
plt.title('Simulated Geometric Brownian Motion Paths')
plt.xlabel("$t$"); plt.ylabel("$S_t$")
plt.savefig("img/GBMpaths.svg"); plt.close()
```

- The compiled code produces the same results:

In [17]:
```
np.random.seed(0)
_, S1 = gbmsim(1, 1000)
_, S2 = gbmsim_vec(1, 1000, seed=0)
np.allclose(S1, S2)
```

Out[17]:
```
True
```

- But it is quite a bit faster:

In [18]:
```
%%timeit #Cell magic (for timing the entire cell).
for k in xrange(10):  #10 paths.
    gbmsim(1, 1000)
```

```
10 loops, best of 3: 23.3 ms per loop
```

In [19]:
```
%timeit gbmsim_vec(1, 1000, numsim=10)
```

```
The slowest run took 553.56 times longer than the fastest. This could mean that an intermediate result is bei
ng cached.
1 loop, best of 3: 570 µs per loop
```

Computational Finance

AMSTERDAM BUSINESS SCHOOL **Business**

Advanced Monte Carlo Methods

Variance Reduction Techniques

- In standard Monte Carlo, the length of the confidence interval for $\theta \equiv \mathbb{E}[X]$ is proportional to $\hat{\sigma}/\sqrt{n}$, where σ is the standard deviation of X.
- Thus to increase the accuracy by a factor of 10 (i.e., gain 1 digit), we need 100 times as many samples.
- Variance reduction techniques aim to improve the accuracy of the estimate, without increasing n.
- We will consider two such techniques: *antithetic sampling*, and *control variates*.
- Another powerful technique is *importance sampling*, but this is beyond the scope of this course.

Antithetic Sampling

- The crude MC estimate for $\theta \equiv \mathbb{E}[X]$, based on n independent draws X_i, is
$$\hat{\theta} \equiv \frac{1}{n}\sum_{i=1}^{n} X_i.$$
- Now suppose that we can somehow sample n pairs $(X_i, \tilde{X}_i)$, such that
  - both X_i and $\tilde{X}_i$ are drawn from the distribution of X;
  - the *pairs* $(X_i, \tilde{X}_i)$ are independent across i;
  - for each i, X_i and $\tilde{X}_i$ are (negatively) correlated.
- The antithetic variable estimator is then
$$\hat{\theta}_{AV} \equiv \frac{1}{2}\left(\frac{1}{n}\sum_{i=1}^{n} X_i + \frac{1}{n}\sum_{i=1}^{n} \tilde{X}_i\right).$$

- Rewriting the estimator as
$$\hat{\theta}_{AV} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{X_i + \tilde{X}_i}{2}\right),$$
we see that it is unbiased, and that $\hat{\theta}_{AV}$ is the mean of n independent observations $(X_i + \tilde{X}_i)/2$, so that the LLN and CLT continue to apply.
- Hence, by the CLT,
$$\sqrt{n}(\hat{\theta}_{AV} - \theta) \xrightarrow{d} \mathrm{N}(0, \sigma_{AV}^2),$$
where
$$\begin{aligned}\sigma_{AV}^2 &\equiv \mathrm{var}\left[\frac{X_i + \tilde{X}_i}{2}\right] = \frac{1}{4}\big(\mathrm{var}[X_i] + \mathrm{var}[\tilde{X}_i] + 2\mathrm{cov}[X_i, \tilde{X}_i]\big)\\ &= \frac{\sigma^2}{2}\big(1 + \rho(X_i, \tilde{X}_i)\big).\end{aligned}$$

- Comparing the variance of $\hat{\theta}_{AV}$, $\frac{1}{n}\sigma_{AV}^2$, with that of the crude estimator based on $2n$ observations $\left(\frac{1}{2n}\sigma^2\right)$, we see that efficiency is gained whenever the correlation $\rho(X_i, \tilde{X}_i)$ is negative.
- So how do we draw correlated random numbers X_i and $\tilde{X}_i$?
- One way is as follows: our simulations are typically based on an array of standard normal random numbers $\mathbf{z}_i$, i.e., $X_i = f(\mathbf{z}_i)$. Setting $\tilde{X}_i = f(-\mathbf{z}_i)$ will often do the trick.
- Note for standard Brownian motion, this corresponds to flipping the path about the abscissa.
- Let's modify last week's `bmsim_vec` and `asianmc_vec` to use antithetic sampling:

In [1]:
```
import numpy as np
from scipy.stats import norm
```

In [2]:
```
def bmsim_vec(T, N, X0=0, mu=0, sigma=1, numsim=1, av=False):
    """Simulate `numsim` Brownian motion paths. If av=True, then 2*`numsim` paths are returned,
    where paths numsim:2numsim+1 are the antithetic paths.
    """
    deltaT = float(T)/N
    tvec = np.linspace(0, T, N+1)
    z = np.random.randn(numsim, N+1)
    if av:
        z = np.concatenate((z, -z))
    dX = mu*deltaT + sigma*np.sqrt(deltaT)*z
    dX[:, 0] = 0.
    X = np.cumsum(dX, axis=1)
    X += X0
    return tvec, X
```


In [3]:
```
def asianmc_vec(S0, K, T, r, sigma, delta, N, numsim=10000, av=False):
    """Monte Carlo price of an arithmetic average Asian call.
    Pass `av=True` to use antithetic sampling.
    """
    X0 = np.log(S0)
    nu = r-delta-.5*sigma**2
    _, X = bmsim_vec(T, N, X0, nu, sigma, numsim, av)
    S = np.exp(X)
    payoffs = np.maximum(S[:, 1:].mean(axis=1)-K, 0.)
    g = np.exp(-r*T)*payoffs
    if av:
        g = .5*(g[:numsim]+g[numsim:])
    C = g.mean(); s = g.std()
    zq = norm.ppf(0.975)
    Cl = C - zq/np.sqrt(numsim)*s
    Cu = C + zq/np.sqrt(numsim)*s
    return C, Cl, Cu
```


- Test:

In [4]:
```
S0 = 11; K = 10; T = 3/12.; r = 0.02; sigma = .3; delta = 0.01; N = 10; numsim=10000
np.random.seed(0)
C, Cl, Cu = asianmc_vec(S0, K, T, r, sigma, delta, N, numsim, av=False)
C, Cu-Cl
```

Out[4]: (1.0927262054551385, 0.03592162508407748)

In [5]:
```
np.random.seed(0)
C, Cl, Cu = asianmc_vec(S0, K, T, r, sigma, delta, N, numsim/2, True)
C, Cu-Cl
```

Out[5]: (1.0824190804865295, 0.012604081385357624)

- Not bad. The new interval is about 1/3 as long, with the same amount of work.
- Timings aren't much different:

In [6]:
```
%timeit asianmc_vec(S0, K, T, r, sigma, delta, N, numsim, False)
```

```
100 loops, best of 3: 6.35 ms per loop
```

In [7]:
```
%timeit asianmc_vec(S0, K, T, r, sigma, delta, N, numsim/2, True)
```

```
100 loops, best of 3: 4.52 ms per loop
```

Control Variates

- Antithetic sampling is a general technique that requires little knowledge about the system being simulated.
- Another variance reduction technique is based on *control variates*. It requires a deeper understanding of the problem, but can yield excellent results.
- Suppose again that we want to estimate $\theta \equiv \mathbb{E}[X]$.
- Further suppose that there is another random variable, Y, correlated with X, with known mean $\mu \equiv \mathbb{E}[Y]$.
- Example: X is the discounted payoff of an option and θ its unknown price. Y is the discounted payoff of a second option which unlike the first, can be priced analytically.

- Suppose we can draw n pairs (X_i, Y_i), i.i.d. across i. The control variate estimator, for a given constant c, is

$$\hat{\theta}_c \equiv \bar{X}_n - c(\bar{Y}_n - \mu).$$

- Intuition: suppose X and Y are positively correlated and $c > 0$. If, due to sampling variation, $\bar{Y}_n > \mu$ for a given set of replications, then it is likely that also $\bar{X}_n > \theta$, so it should be corrected downwards.
- The estimator is unbiased:

$$\mathbb{E}[\hat{\theta}_c] = \mathbb{E}[\bar{X}_n] - c\mathbb{E}[\bar{Y}_n - \mu] = \theta. \qquad (\dagger)$$

- Let $Z_i \equiv X_i - c(Y_i - \mu)$. Then $\hat{\theta}_c = n^{-1} \sum_{i=1}^{n} Z_i$, a sum of n i.i.d. terms, so the LLN and CLT apply. Thus

$$\sqrt{n}(\hat{\theta}_c - \theta) \xrightarrow{d} \mathrm{N}(0, \sigma^2_{CV}),$$

where $\sigma^2_{CV} \equiv \mathrm{var}(Z_i)$.

- The variance is seen to be

$$\begin{aligned}\sigma^2_{CV} &\equiv \mathrm{var}(Z_i) = \mathrm{var}\Big(X_i - c(Y_i - \mu)\Big) \\ &= \sigma^2 - 2c\ \mathrm{cov}(X, Y) + c^2 \mathrm{var}(Y).\end{aligned}$$

- It remains to choose c. The optimal c minimizes σ^2_{CV}, yielding the FOC

$$0 = -2\mathrm{cov}(X, Y) + 2c^* \mathrm{var}(Y) \quad \Leftrightarrow \quad c^* = \frac{\mathrm{cov}(X, Y)}{\mathrm{var}(Y)}.$$

- With this choice,

$$\begin{aligned}\sigma^2_{CV} &= \sigma^2 - 2\frac{\mathrm{cov}^2(X, Y)}{\mathrm{var}(Y)} + \frac{\mathrm{cov}^2(X, Y)}{\mathrm{var}(Y)} \Leftrightarrow \\ \frac{\sigma^2_{CV}}{\sigma^2} &= 1 - \rho^2(X, Y).\end{aligned}$$

- We see that the variance is reduced whenever $\rho(X, Y) \neq 0$.
- In practice $c^* = \mathrm{cov}(X, Y)/\mathrm{var}(Y)$ is unknown. A consistent estimator is $\hat{c} \equiv S_{X,Y}/S^2_Y$, which can be obtained by regressing X on Y and a constant.
- $\hat{c}$ is random and not independent of Y. In view of $(\dagger)$, this introduces a bias in $\hat{\theta}_{\hat{c}}$, because we cannot take it out of the expectation. Usually this bias is small (and disappears asymptotically).
- The standard errors (and thus the CI) would also need to be adjusted. We just ignore these difficulties.
- The usefulness of the method hinges on the availability of good control variates. A good control variate has $|\rho(X, Y)|$ close to 1, i.e., a strong linear relationship with X. It is possible to incorporate several control variates.
- Control variates and antithetic sampling can be combined.

- Classic example: using the geometric average asian call (which can be priced analytically) as a control variate for an arithmetic average call (which cannot).
- The geometric average call has payoff

$$C_T^{\text{Geo}} = \left(\prod_{i=1}^{N} S_{t_i}^{1/N} - K \right)^+ = \left(\exp\left\{ \frac{1}{N} \sum_{i=1}^{N} X_{t_i} \right\} - K \right)^+, \qquad X_t \equiv \log S_t.$$

- Assuming that $t_i = i\delta t$, it can be shown that

$$C_0^{\text{Geo}} = e^{(\hat{\mu}-r)T} S_0 \Phi(\hat{d}_1) - e^{-rT} K \Phi(\hat{d}_2),$$

where

$$\hat{d}_1 \equiv \frac{\log(S_0/K) + (\hat{\mu} + \frac{1}{2}\hat{\sigma}^2)T}{\hat{\sigma}\sqrt{T}}, \qquad \hat{d}_2 \equiv \hat{d}_1 - \hat{\sigma}\sqrt{T},$$

$$\hat{\sigma} \equiv \sigma \sqrt{\frac{(N+1)(2N+1)}{6N^2}}, \quad \text{and} \quad \hat{\mu} \equiv \frac{\hat{\sigma}^2}{2} + \left(r - \delta - \frac{\sigma^2}{2} \right) \frac{N+1}{2N},$$

- This is valid only at $t = 0$; the general formula is more complicated.

In [8]:
```
def asiancall_geo(S0, K, T, r, sigma, delta, N):
    """Price of a geometric average price asian call."""
    shat = sigma * np.sqrt(((N+1) * (2*N+1)) / (6.0 * N**2))
    mhat = shat**2/2.0 + (r-delta-.5*sigma**2) * (N+1)/(2.0*N)
    d1 = (np.log(S0/float(K)) + T*(mhat + 0.5*shat**2)) / shat / np.sqrt(T)
    d2 = d1-shat*np.sqrt(T)
    Price = np.exp((mhat-r)*T)*S0*norm.cdf(d1)-np.exp(-r*T)*K*norm.cdf(d2)
    return Price
```


In [9]:
```
def asianmc_cv(S0, K, T, r, sigma, delta, N, numsim=10000):
    """Monte Carlo price of an arithmetic average Asian call using control variates.
    """
    X0 = np.log(S0)
    nu = r-delta-.5*sigma**2
    _, X = bmsim_vec(T, N, X0, nu, sigma, numsim)
    S = np.exp(X)
    mu = asiancall_geo(S0, K, T, r, sigma, delta, N)
    payoffs = np.maximum(S[:, 1:].mean(axis=1)-K, 0.)
    control = np.maximum(np.exp(X[:, 1:].mean(axis=1))-K, 0.)
    c = (np.cov(payoffs, control)/np.var(control))[0, 1]
    g = np.exp(-r*T)*payoffs - c*(np.exp(-r*T)*control-mu)
    C = g.mean(); s = g.std()
    zq = norm.ppf(0.975)
    Cl = C - zq/np.sqrt(numsim)*s
    Cu = C + zq/np.sqrt(numsim)*s
    return C, Cl, Cu
```


- Let's try it out:

In [10]:
```
np.random.seed(0)
C, Cl, Cu = asianmc_cv(S0, K, T, r, sigma, delta, 10, numsim)
C, Cu-Cl
```

Out[10]: (1.0835250424516814, 0.00060646107619888312)

- Wow. Obviously the two payoffs are very highly correlated.
- Timings:

In [11]:
```
%timeit asianmc_vec(S0, K, T, r, sigma, delta, N, numsim)
```

```
100 loops, best of 3: 5.98 ms per loop
```

In [12]:
```
%timeit asianmc_cv(S0, K, T, r, sigma, delta, N, numsim)
```

```
100 loops, best of 3: 6.64 ms per loop
```

The Expat Conundrum: Greeks in Monte Carlo

- So far we were mostly concerned with *pricing* options. An equally important problem in practice is *hedging*: suppose that a financial institution has issued a call option (i.e., it is short the option). In order to eliminate the risk, it might create the hedge
$$\Pi_t \equiv -C_t + \phi_t S_t,$$
where ϕ_t is a number of stocks. The *delta-hedged* portfolio has
$$\phi_t = \Delta_t \equiv \frac{\partial C_t}{\partial S_t}.$$
- This position is *delta-neutral*: it is immune to (infinitesimally) small movements of the underlying.
- The portfolio has to be rebalanced every time the underlying moves (because Δ_t depends on time), incurring trading costs.

- If there is another instrument on the same underlying available, then it is possible to construct a portfolio that is also *gamma-neutral*. Gamma is defined as
$$\Gamma_t \equiv \frac{\partial^2 C_t}{\partial S_t^2}.$$
Such a portfolio may need to be readjusted less frequently.
- Δ_t and Γ_t are examples of the so-called option *Greeks*. Other examples are θ_t (the derivative with respect to time), ρ_t (w.r.t. r), and $\mathcal{V}_t$ ("Vega", w.r.t. σ).
- In the BS model, the latter two parameters are constant, but in practice they are not.
- In order to construct a hedge, it is important that we be able to compute the Greeks. This is easy if we have an analytical expression for the price; e.g., for a European call in the BS model, it can be shown that
$$\Delta_t = e^{-\delta(T-t)}\Phi(d_1).$$
- Chapter 18 of Hull (2012) lists the expressions for the other Greeks.

- If the option cannot be priced analytically, then we may need to resort to Monte Carlo.
- Recalling our risk neutral pricing formula, we see that
$$\Delta_0 = \frac{\partial}{\partial S_0}\mathbb{E}^{\mathbb{Q}}\left[e^{-rT}C_T(S_0)\right].$$
- In view of the definition of the derivative,
$$\Delta_0 \equiv \lim_{\delta S_0 \to 0} \frac{\mathbb{E}^{\mathbb{Q}}\left[e^{-rT}C_T(S_0+\delta S_0)\right] - \mathbb{E}^{\mathbb{Q}}\left[e^{-rT}C_T(S_0)\right]}{\delta S_0},$$
it is natural to approximate Δ_0 with a finite difference quotient for some small δS_0:

In [13]:
```
dS = .01
np.random.seed(0)
Cd, _, _ = asianmc_vec(S0+dS, K, T, r, sigma, delta, N, numsim)
C,  _, _ = asianmc_vec(S0,    K, T, r, sigma, delta, N, numsim)
Delta = (Cd-C)/dS; Delta
```

Out[13]:
```
1.5653871014334575
```

- The true answer is around 0.858.

- Let's try to improve the approximation by reducing δS_0:

In [14]:
```
dS = .001; np.random.seed(0)
Cd, _, _ = asianmc_vec(S0+dS, K, T, r, sigma, delta, N, numsim)
C,  _, _ = asianmc_vec(S0,    K, T, r, sigma, delta, N, numsim)
Delta = (Cd-C)/dS; Delta
```

Out[14]:
```
7.886858496600313
```

- Ouch. What's happening is that as $\delta S_0 \to 0$, the sampling variation in C_T increasingly dominates the variation from a small perturbation of S_0. This increases the variance of the estimator (which is nevertheless unbiased as $\delta S_0 \to 0$).
- The trick is to use the same random numbers in both simulation runs:

In [15]:
```
dS = .001; np.random.seed(0)
Cd, _, _ = asianmc_vec(S0+dS, K, T, r, sigma, delta, N, numsim)
np.random.seed(0)
C,  _, _ = asianmc_vec(S0,    K, T, r, sigma, delta, N, numsim)
Delta = (Cd-C)/dS; Delta
```

Out[15]:
```
0.8622451045858792 2
```

- Essentially, this *method of common random numbers* amounts to the approximation

$$\Delta_0 \approx \mathbb{E}^{\mathbb{Q}}\left[\frac{e^{-rT}C_T(S_0+\delta S_0)-e^{-rT}C_T(S_0)}{\delta S_0}\right].$$

- The validity of this approach hinges on whether it is justified to take the limit in the above expression, i.e., whether we may exchange the order of derivative and expectation so that

$$\frac{\partial}{\partial S_0}\mathbb{E}^{\mathbb{Q}}\left[e^{-rT}C_T(S_0)\right] = \mathbb{E}^{\mathbb{Q}}\left[\frac{\partial}{\partial S_0}e^{-rT}C_T(S_0)\right].$$

- The approach can be made more explicit. The random variable of interest is

$$e^{-rT}C_T = e^{-rT}(S_T-K)^+,$$

where

$$S_T = S_0 e^{(r-\delta-\sigma^2/2)T+\sigma\sqrt{T}Z}, \quad Z\sim N(0,1).$$

- By the chain rule,

$$\frac{\partial C_T}{\partial S_0} = \frac{\partial C_T}{\partial S_T}\frac{\partial S_T}{\partial S_0}.$$

- We have

$$\frac{\partial S_T}{\partial S_0} = \frac{S_T}{S_0} \quad \text{and} \quad \frac{\partial C_T}{\partial S_T} = \begin{cases} 0, & \text{if } S_T < K \\ 1, & \text{if } S_T > K \end{cases}.$$

- The latter is undefined at $S_T = K$, but this happens with zero probability. Hence

$$\frac{\partial C_T}{\partial S_0} = \frac{S_T}{S_0}\mathbf{1}_{S_T>K}. \qquad (\ddagger)$$

- Thus, provided the exchange of derivative and expectation is valid, we have

$$\Delta_0 = \mathbb{E}^{\mathbb{Q}}\left[e^{-rT}\frac{S_T}{S_0}\mathbf{1}_{S_T>K}\right].$$

- This is called the *pathwise estimate*, because we are keeping the asset path (the random numbers) the same when taking differences / derivatives.
- A similar derivation shows that for the arithmetic average asian call,

$$\Delta_0 = \mathbb{E}^{\mathbb{Q}}\left[e^{-rT}\frac{\bar{S}_T}{S_0}\mathbf{1}_{\bar{S}_T>K}\right].$$

This is genuinely useful, as no closed form expression exists.

- Notes:
  - Unlike the simple common-random-numbers approach, the explicit pathwise approach allows us to construct a CI.
  - Also, antithetic sampling and control variates can be used, just like for prices.
  - The validity of the method depends on whether the exchange of derivative and expectation is valid; a necessary condition for this is continuity of the payoff function. Thus, the method fails for, e.g., binary options (note that there, $\partial C_T/\partial S_T = 0$ almost everywhere).
  - This also implies that the method cannot be applied to second derivatives such as the option gamma; observe that ($\ddagger$) is discontinuous. Alternatives include the *likelihood ratio method* (which we won't discuss).

- We apply the method to the arithmetic average Asian option:

In [16]:
```
def asianmc_delta(S0, K, T, r, sigma, delta, N, numsim=10000):
    """Price and Delta of an arithmetic average Asian call.
    """
    X0 = np.log(S0); nu = r-delta-.5*sigma**2
    _, X = bmsim_vec(T, N, X0, nu, sigma, numsim)
    S = np.exp(X)
    payoffs = np.maximum(S[:, 1:].mean(axis=1)-K, 0.)
    deltas = (payoffs > 0)/float(S0)*S[:, 1:].mean(axis=1)
    g1 = np.exp(-r*T)*payoffs
    g2 = np.exp(-r*T)*deltas
    C = g1.mean(); s1 = g1.std()
    zq = norm.ppf(0.975)
    Cl = C - zq/np.sqrt(numsim)*s1
    Cu = C + zq/np.sqrt(numsim)*s1
    Delta = g2.mean(); s2 = g2.std()
    Dl = Delta - zq/np.sqrt(numsim)*s2
    Du = Delta + zq/np.sqrt(numsim)*s2
    return C, Cl, Cu, Delta, Dl, Du
```

In [17]:
```
np.random.seed(0)
C, Cl, Cu, Delta, Dl, Du = asianmc_delta(S0, K, T, r, sigma, delta, N)
print(C, Cl, Cu, Delta, Dl, Du)
```

```
(1.0927262054551385, 1.0747653929130998, 1.1106870179971773, 0.86224195045147467, 0.85482635381446825, 0.8696
5754708848109)
```


- Finally, we combine the pathwise estimate with antithetic sampling and a control variate.
- We'll use the delta of the geometric average call as a control variate. The first step is to modify `asiancall_geo` to also return the delta of the option:

In [18]:
```
def asiancall_geo2(S0, K, T, r, sigma, delta, N):
    """Price and Delta of a geometric average price asian call.
    """
    shat = sigma * np.sqrt(((N+1) * (2*N+1)) / (6.0 * N**2))
    mhat = shat**2/2.0 + (r-delta-.5*sigma**2) * (N+1)/(2.0*N)
    d1 = (np.log(S0/float(K)) + T*(mhat + 0.5*shat**2)) / shat / np.sqrt(T)
    d2 = d1-shat*np.sqrt(T)
    Price = np.exp((mhat-r)*T)*S0*norm.cdf(d1)-np.exp(-r*T)*K*norm.cdf(d2)
    Delta = np.exp((mhat-r)*T)*norm.cdf(d1)
    return Price, Delta
```

- We'll name the final version of our code `asianMC`.
- To make it easier to use, it returns a named tuple containing the simulation results, instead of a bunch of scalars.

In [19]:
```
from collections import namedtuple
MonteCarloResults = namedtuple("MonteCarloResults", ["Price", "Cl", "Cu", "Delta", "Dl", "Du"])
def asianMC(S0, K, T, r, sigma, delta, N, numsim=10000, av=True):
    """Price and Delta of an arithmetic average Asian call using control variates.
    Pass `av=False` to not use antithetic sampling.
    """
    X0 = np.log(S0); nu = r-delta-.5*sigma**2
    _, X = bmsim_vec(T, N, X0, nu, sigma, numsim, av)
    S = np.exp(X)
    mu, D = asiancall_geo2(S0, K, T, r, sigma, delta, N)
    payoffs = np.maximum(S[:, 1:].mean(axis=1)-K, 0.)
    control1 = np.maximum(np.exp(X[:, 1:].mean(axis=1))-K, 0.)
    deltas = (payoffs > 0)/float(S0)*S[:, 1:].mean(axis=1)
    control2 = (control1 > 0)/float(S0)*np.exp(X[:, 1:].mean(axis=1))
    c1 = (np.cov(payoffs, control1)/np.var(control1))[0, 1]
    c2 = (np.cov(deltas, control2)/np.var(control2))[0, 1]
    g1 = np.exp(-r*T)*payoffs - c1*(np.exp(-r*T)*control1-mu)
    g2 = np.exp(-r*T)*deltas - c2*(np.exp(-r*T)*control2-D)
    if av:
        g1 = .5*(g1[:numsim]+g1[numsim:])
        g2 = .5*(g2[:numsim]+g2[numsim:])
    C = g1.mean(); s1 = g1.std(); zq = norm.ppf(0.975)
    Cl = C - zq/np.sqrt(numsim)*s1
    Cu = C + zq/np.sqrt(numsim)*s1
    Delta = g2.mean(); s2 = g2.std()
    Dl = Delta - zq/np.sqrt(numsim)*s2
    Du = Delta + zq/np.sqrt(numsim)*s2
    return MonteCarloResults(C, Cl, Cu, Delta, Dl, Du)
```


In [20]:
```
np.random.seed(0)
results=asianMC(S0, K, T, r, sigma, delta, N)
results.Price, results.Delta
```

Out[20]:
```
(1.0834462969035707, 0.85870108333211115)
```